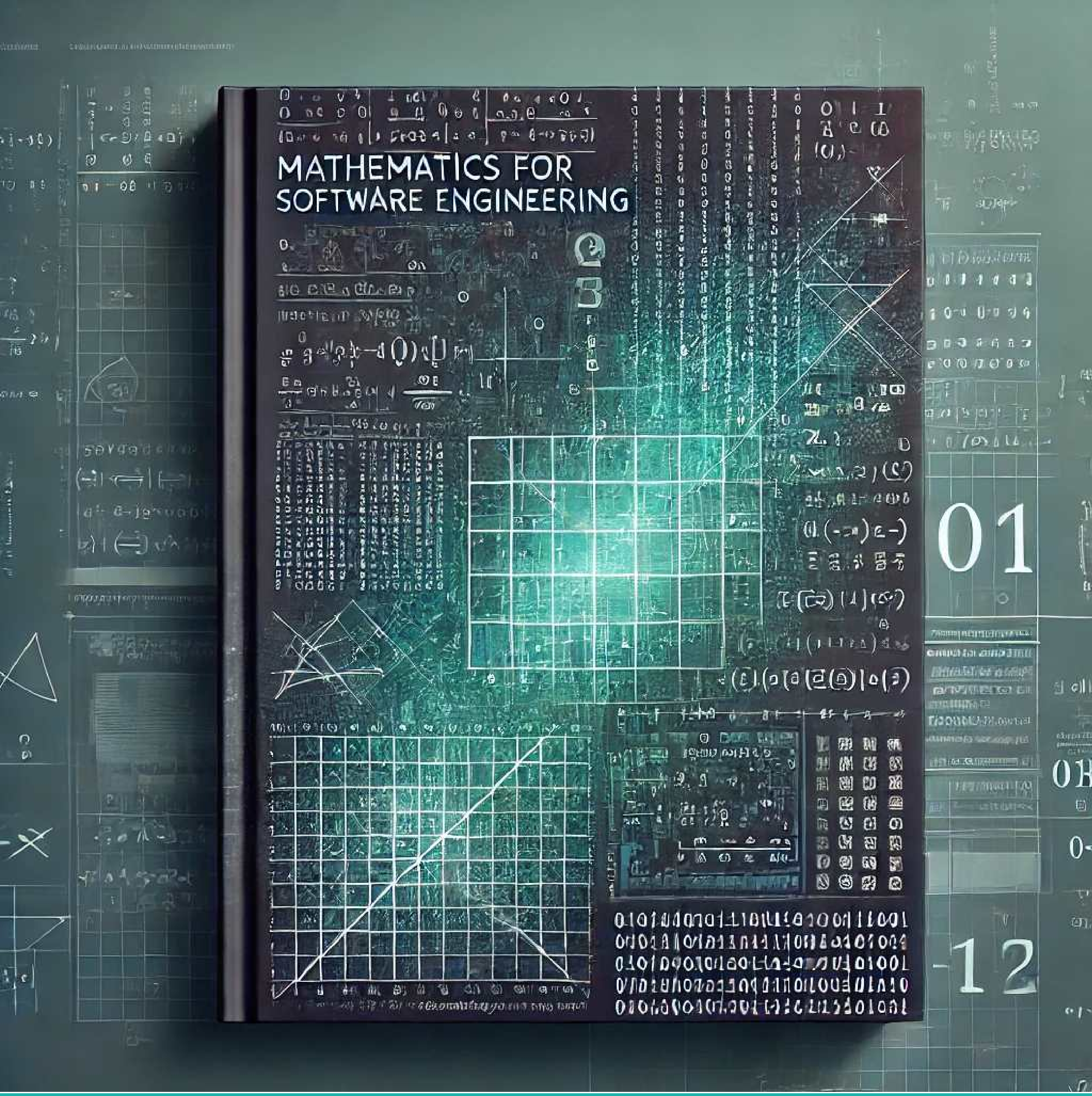


Mathematics for Software Engineering

Authors: Richard Brooks & Eduard Fekete

Date: May, 2025

Version: 2.0

Contents

Chapter 1 Basic Arithmetic and Functions **1**

1.1 Factorisation and the Order of Operations . . . 1
1.2 Fractions . . . 3
1.3 Exponents, Radicals and Surds . . . 5
1.4 Using Formulae and Substitution . . . 9
1.5 Rearranging Formulae . . . 10
1.6 Functions . . . 12
1.7 Graphical Identification of Function Types . . . 17
1.8 Logarithms . . . 21

Chapter 2 Fundamental Concepts in Number Theory **28**

2.1 Types of Numbers . . . 29
2.2 Integer Properties and Modular Arithmetic . . . 33

Chapter 3 Numeral Systems **38**

3.1 Binary Numbers . . . 38
3.2 Octal and Hexadecimal . . . 49
3.3 Converting Between Systems . . . 50

Chapter 4 Boolean Algebra **55**

4.1 Introduction to Boolean Algebra . . . 55
4.2 Basic Operations on Bits . . . 57
4.3 Boolean Functions . . . 59
4.4 Logic Gates . . . 64

Chapter 5 Combinatorics and Probability Theory **67**

5.1 Sample Space and Events . . . 67
5.2 Types of Events in Probability Theory . . . 69
5.3 Counting Principles . . . 72
5.4 Probability Basics . . . 77
5.5 Probability of Joint Events and Set Operations . . . 79

Chapter 6 Conditional Probability and Bayes' Theorem **83**

6.1 Conditional Probability . . . 83
6.2 Multiplication and Total Probability Rules . . . 88
6.3 Independence . . . 91
6.4 Bayes' Theorem . . . 94

Chapter 7 Linear Equations in Linear Algebra **97**

7.1 Systems of Linear Equations . . . 97
7.2 Elementary Row Operations and Echelon Forms . . . 103

Chapter 8 Vectors and Matrices **108**

8.1 Vectors in $\mathbb{R}^2$ and $\mathbb{R}^n$ 108

8.2 Matrix Equations 113

8.3 Solution Sets of Linear Systems 117

8.4 Linear Independence 121

Chapter 9 Matrix Algebra **125**

9.1 Matrix Arithmetic 125

9.2 Matrix Transpose 131

9.3 Invertible Matrices 133

9.4 The Invertible Matrix Theorem 139

Chapter 10 Time Complexity **141**

10.1 Introduction to Complexity Analysis 141

10.2 Growth of Functions 142

10.3 Asymptotic Analysis 146

10.4 Further Analysis 152

Chapter 11 Code Analysis Techniques **161**

11.1 Pseudocode 162

11.2 Construct Analysis 166

11.3 Loop Analysis 176

11.4 Code Analysis 188

Bibliography **197**

Appendix A Summation **198**

Chapter 1 Basic Arithmetic and Functions

In the study of mathematics, a solid understanding of basic arithmetic operations and the concept of functions forms the foundation for more advanced topics. Arithmetic involves the manipulation of numbers through fundamental operations such as addition, subtraction, multiplication, and division. These operations are not only essential in everyday calculations but also serve as the building blocks for more complex mathematical procedures.

Functions, on the other hand, represent a crucial concept in mathematics, serving as a bridge between arithmetic and higher-level mathematical analysis. A function can be thought of as a special relationship between two sets, where each input (from the domain) is associated with exactly one output (from the co-domain). Understanding functions and their properties allows us to model and solve real-world problems with greater precision and flexibility.

In this section, we will explore the basic arithmetic rules and introduce the concept of functions, including their definitions, notations, and key properties. We will also discuss how these concepts are applied in various contexts, setting the stage for more advanced mathematical discussions.

1.1 Factorisation and the Order of Operations

Recall that when computing a product such as

$$a(b + c) = ab + ac$$

we distribute the factor a across each term inside the parentheses. We call this **the distributive property**. However, it is often advantageous or necessary to perform the reverse operation, known as factorisation. Factorisation involves expressing the sum $ab + ac$ in its factored form $a(b + c)$.

Mathematically, the expression $a(b+c)$ is considered more simplified or "better" than $ab+ac$. To understand why, we must distinguish between **terms** and **factors**. Terms are separated by addition or subtraction, while factors are separated by multiplication or division. For instance, the expression

$$8 - 5 + 3$$

consists of three terms (8, -5, and 3), whereas the expression

$$2 \times 5 \times 3$$

consists of three factors (2, 5, and 3). Consider the expression

$$5 + 2 + 7 \times 8$$

which consists of three terms (5, 2, and 7×8), and note that one of the terms itself contains two factors (7 and 8). Similarly, the expression

$$5(2 + 4)(-9)$$

consists of three factors (5, $2 + 4$, and -9), with one factor, $2 + 4$, containing two terms.

The advantage of expressing mathematical expressions solely in terms of factors lies in the ability to simplify them more effectively. This concept will be elaborated throughout this chapter, especially when dealing with fractions and equations. Consider the following examples of factorisation:

$$8 - 5 + 3$$

consists of three terms (8, -5 and 3), while the expression

$$2 \times 5 \times 3$$

consists of three factors (2, 5 and 3). The expression

$$5 + 2 + 7 \times 8$$

consists of three terms (5, 2 and 7 × 8) and one of the terms consists of two factors (7 × 8), while the expression

$$5(2 + 4)(-9)$$

consists of three factors (5, (2 + 4), (-9)) and one of the factors consists of two terms ((2 + 4)).

It can be advantageous to express terms solely in factors because this allows us to simplify the expressions. This will become clearer as we progress through the lesson, and it is particularly important when working with fractions and equations. Here are some more examples of factorisation:

Example 1.1 Factorisation

$$\begin{aligned}
ab - ac &= a(b - c) \\
-ab - ac &= a(-b - c) \\
-ab - ac &= -a(b + c) \\
ab - ac + aa - aa &= a(b - c + a - a) \\
abc - ab + aba &= ab(c - 1 + a) \\
ab - abc - a &= a(b - bc - 1) = b(1 - c) - 1
\end{aligned}$$

When evaluating mathematical expressions, it is crucial to follow a specific order of operations to ensure accurate results. The correct sequence for performing these operations is as follows:

1. **Brackets (Parentheses)**: First, perform all operations inside brackets or parentheses.
2. **Exponents and Radicals**: Next, evaluate exponents (powers) and radicals (roots).
3. **Multiplication and Division**: Then, perform multiplication and division from left to right as they appear.
4. **Addition and Subtraction**: Finally, execute addition and subtraction from left to right as they appear.

Let's consider examples for each operation to illustrate the order of operations:

Example 1.2 Brackets
Evaluate the expression: $(2 + 3) \times 4$

$$(2 + 3) \times 4 = 5 \times 4 = 20$$

Example 1.3 Exponents and Radicals
Evaluate the expression: $3^2 + \sqrt{16}$

$$3^2 + \sqrt{16} = 9 + 4 = 13$$

Example 1.4 Multiplication and Division
Evaluate the expression: $6 \div 2 \times 3$. According to the standard order of operations, we perform division and multiplication from left to right:

$$6 \div 2 \times 3 = (6 \div 2) \times 3 = 3 \times 3 = 9$$

Example 1.5 Addition and Subtraction
Evaluate the expression: $8 - 3 + 2$

$$8 - 3 + 2 = 5 + 2 = 7$$

1.2 Fractions

By definition, a fraction always consists of (at least) two factors. The first factor we will call the **numerator** and is the "top part" of the fraction. The bottom part we will call the **denominator**. Perhaps the most important rule when working with fractions is that two fractions can only be added or subtracted if they have identical denominators. Also, the denominator must never be equal to 0.

Example 1.6

$$\frac{1}{x^2 - 2}$$

Here we must make sure that $x^2 - 2 \neq 0$ which means that we may only use values different from $\pm\sqrt{2}$.

Rules for Calculations Involving Fractions

(1)	$\frac{a}{b} \times m = \frac{am}{b}$	where $b \neq 0$
(2)	$\frac{a}{b} \div m = \frac{a}{bm}$	where $b \neq 0$ and $m \neq 0$
(3)	$m \div \frac{a}{b} = m \times \frac{b}{a} = \frac{mb}{a}$	where $b \neq 0$ and $a \neq 0$
(4)	$\frac{a}{b} \times \frac{c}{a} = \frac{ac}{ba}$	where $b \neq 0$ and $a \neq 0$
(5)	$\frac{a}{b} \div \frac{c}{a} = \frac{a}{b} \times \frac{a}{c}$	where $b \neq 0$ and $a \neq 0$
(6)	$\frac{a}{b} = \frac{ac}{bc}$	where $b \neq 0$ and $c \neq 0$
(7)	$\frac{a}{b} + \frac{c}{a} = \frac{aa}{ba} + \frac{cb}{ba} = \frac{aa + cb}{ba}$	where $b \neq 0$ and $a \neq 0$

To extend or reduce a fraction, we must multiply or divide by the same numbers in the denominator and numerator:

Example 1.7 Extending or reducing fractions

$$\frac{72}{144} = \frac{72 \div 12}{144 \div 12} = \frac{6}{12} = \frac{6 \div 6}{12 \div 6} = \frac{1}{2}$$

$$\frac{2}{3}=\frac{2\times 6}{3\times 6}=\frac{12}{18}$$
$$\frac{2x}{2xx}=\frac{2}{2x}$$
$$\frac{3a}{6a+3b}=\frac{3a}{3(2a+b)}=\frac{a}{2a+b}$$

To factorise an expression, all terms must be divided or multiplied uniformly. This implies that it is not possible to simplify the following expression any further, even though it might be tempting:

$$\frac{a}{2a+b}\neq\frac{1}{2+b}$$

For proper factorisation of $\frac{a}{2a+b}$, you must divide a into all terms in the denominator:

$$\frac{a}{2a+b}=\frac{1}{2+\frac{b}{a}}$$

As illustrated above, the multiplication of fractions is straightforward: you multiply the numerators and denominators with each other, respectively.

Example 1.8 Multiplication of fractions

Consider the fractions $\frac{a}{b}$ and $\frac{c}{d}$. Their product is:

$$\frac{a}{b}\times\frac{c}{d}=\frac{a\times c}{b\times d}$$

For instance, if $a=2$, $b=3$, $c=4$, and $d=5$, then:

$$\frac{2}{3}\times\frac{4}{5}=\frac{2\times 4}{3\times 5}=\frac{8}{15}$$

Example 1.9 Dividing fractions

$$\frac{6}{7}\div\frac{4}{21}=\frac{6}{7}\times\frac{21}{4}=\frac{126}{28}=\frac{63}{14}=\frac{9}{2}$$
$$\frac{1}{2}\div 3=\frac{1}{2}\div\frac{3}{1}=\frac{1}{2}\times\frac{1}{3}=\frac{1}{6}$$
$$\frac{2}{3}\div\frac{8}{9}=\frac{2}{3}\times\frac{9}{8}=\frac{18}{24}=\frac{3}{4}$$
$$\frac{8}{9}\div 16=\frac{8}{9}\times\frac{1}{16}=\frac{8}{144}=\frac{4}{72}=\frac{1}{18}$$

Adding and subtracting fractions seems to cause more problems than multiplication and division. The key is to find a common denominator between the fractions and then remember the above-mentioned rule about extending fractions.

Example 1.10 Adding and subtracting fractions

$$\frac{1}{5}+\frac{2}{5}=\frac{1+2}{5}=\frac{3}{5}$$
$$\frac{1}{4}+\frac{2}{3}=\frac{1\times 3}{4\times 3}+\frac{2\times 4}{3\times 4}=\frac{3}{12}+\frac{8}{12}=\frac{3+8}{12}=\frac{11}{12}$$
$$\frac{7}{12}-\frac{5}{8}=\frac{7\times 8}{12\times 8}-\frac{5\times 12}{8\times 12}=\frac{56}{96}-\frac{60}{96}=\frac{56-60}{96}=\frac{-4}{96}=\frac{-1}{24}$$

Note: Never do

$$\frac{a}{b}+\frac{c}{d}=\frac{a+b}{b+d}$$

Example 1.11

$$\frac{x}{x-1} \times \frac{2}{x(x+4)} = \frac{2x}{(x-1)x(x+4)} = \frac{2}{(x-1)(x+4)}$$

$$\frac{2}{x-1} \div \frac{x}{x-1} = \frac{2}{x-1} \times \frac{x-1}{x} = \frac{2(x-1)}{(x-1)x} = \frac{2}{x}$$

$$\frac{x+1}{x^2+2} + \frac{x-6}{x^2+2} = \frac{x+1+x-6}{x^2+2} = \frac{2x-5}{x^2+2}$$

Remember, when a fraction is preceded by a minus sign, all signs in the numerator must be changed accordingly. This is similar to how you would change all signs within parentheses when they are preceded by a minus sign.

Consider the expression:

$$-\frac{a-b}{c}$$

To correctly handle the negative sign, change all the signs in the numerator:

$$-\frac{a-b}{c} = \frac{-a+b}{c}$$

For example, if $a = 5$ and $b = 3$, then:

$$-\frac{5-3}{c} = \frac{-5+3}{c} = \frac{-2}{c}$$

Example 1.12

$$\frac{x+1}{x^2+2} - \frac{x-6}{x^2+2} = \frac{x+1-x+6}{x^2+2} = \frac{7}{x^2+2}$$

1.3 Exponents, Radicals and Surds

An **exponent** is a shortcut for repeated multiplication of the same number:

Example 1.13 Exponentiation

$$4 \times 4 \times 4 \times 4 \times 4 = 4^5$$

$$x \times x \times x \times x \times x = x^5$$

Radicals, or **roots**, represent the inverse operation of applying exponents. A radical is any number expressed with the radical symbol $\sqrt{}$. Specifically, applying a radical can reverse the effect of an exponent, and vice versa. For instance, squaring 2 yields 4, and taking the square root of 4 returns 2. Similarly, squaring 3 results in 9, and the square root of 9 brings us back to 3.

Example 1.14 Taking the root

$$\sqrt{a} \times \sqrt{a} = (\sqrt{a})^2 = a$$

$$\sqrt{a} = b \implies (\sqrt{a})^2 = b^2 \iff a = b^2$$

A **surd** is a type of radical that is both real and irrational, examples include $\sqrt{2}$, $\sqrt{3}$, $\sqrt{5}$, and $\sqrt{6}$.

Numbers can be raised to powers other than 2, such as cubing (raising to the third power), or even raising to the fourth power, the 100th power, and so forth. Correspondingly, you can take the cube root of a number, the fourth root, the 100th root, and so on. To indicate a root other than a square root, the same radical symbol is used, but with a number called the index inserted into the radical sign, typically positioned within the "check mark" part.

Example 1.15 Index and argument

$$4^3 = 64 \iff \sqrt[3]{64} = 4$$

In this example, the "3" inside the radical sign is the **index** of the radical. The "64" is referred to as the argument of the **radical**, also known as the **radicand**. Since square roots are the most common type of radicals, the index is usually omitted for square roots. Although "$\sqrt[2]{2}$" would be technically correct, it is rarely used in practice.

Rules for calculations involving radicals

(1)	$\sqrt[n]{xy} = \sqrt[n]{x} \times \sqrt[n]{y}$	where $x, y \geq 0$
(2)	$\sqrt[n]{\frac{x}{y}} = \frac{\sqrt[n]{x}}{\sqrt[n]{y}}$	where $x \geq 0$ and $y > 0$
(3)	$\sqrt{x^2} = \lvert x \rvert$	where $x \in \mathbb{R}$
(4)	$(\sqrt[n]{x})^n = x$	If $x < 0$ and $n \in \mathbb{N}$, then $\sqrt[n]{x}$ is not defined
(5)	$\sqrt[n]{-x} = -\sqrt[n]{x}$	where $x \geq 0$ and $n \in \mathbb{N}$ is odd

Raising a number to a **power**, also known as **exponentiation**, is a fundamental mathematical operation that involves multiplying a number by itself a certain number of times as we saw above. The **base** is the number being multiplied, and the **exponent** indicates how many times the base is used as a factor. For example, a^n means that the base a is multiplied by itself n times. Exponentiation is a powerful tool in mathematics, with a few essential rules that govern its application.

Rules for calculations involving exponents 1

Let $n, m \in \mathbb{N}$. Then the following applies:

(1)	$x^n \cdot x^m = x^{n+m}$	
(2)	$\frac{x^n}{x^m} = x^{n-m}$	$x \neq 0$
(3)	$x^n \cdot y^n = (x \cdot y)^n$	
(4)	$\frac{x^n}{y^n} = \left(\frac{x}{y}\right)^n$	$y \neq 0$
(5)	$(x^n)^m = x^{n \cdot m}$	
(6)	$x^1 = x$	

Some of these rules allow the concept of powers to be extended so that the exponent may be any integer. If you set $n = m$ in rule (2), you get:

$$\frac{x^n}{x^n} = x^{n-n} = x^0$$

But since $\frac{x^n}{x^n} = 1$, we obtain

$$x^0 = 1$$

Thus, the concept of exponentiation is extended to include $n \in \mathbb{N} \cup \{0\}$. If you now set $n = 0$ again in rule (2), you get:

$$\frac{x^0}{x^m} = x^{0-m} = x^{-m}$$

But according to the previous calculation, $x^0 = 1$. Therefore, you obtain

$$\frac{1}{x^m} = x^{-m}$$

Since m is a positive number, $-m$ must be a negative number. Thus, the concept of exponentiation is extended to apply to all integers. This means that definition our concept of powers holds for $n \in \mathbb{Z}$ and that the rules in definition 1.3 apply for $n \in \mathbb{Z}$ and $m \in \mathbb{Z}$. As a consequence, the following rules can be added:

Rules for calculations involving exponents 2

Let $n \in \mathbb{Z}$. Then the following applies:

(7) $\qquad x^0 = 1 \qquad x \neq 0$

(8) $\qquad \frac{1}{x^m} = x^{-m} \qquad x \neq 0$

Let us illustrate these rules with a couple of examples.

Example 1.16 Reduce the following expression

$$\left(3xy^6\right)^3 = 3^3 \cdot x^3 \cdot \left(y^6\right)^3 = 27x^3y^{18}$$

Example 1.17 Reduce the following expression

$$\frac{a^{-4}b^3}{a^7b^{-5}} = \frac{a^{-4}}{a^7} \cdot \frac{b^3}{b^{-5}} = a^{-4-7} \cdot b^{3-(-5)} = a^{-11} \cdot b^8 = \frac{b^8}{a^{11}}$$

For positive numbers x, the concept of exponentiation can be further extended to apply when the exponent is a rational number. Any rational number $r \in \mathbb{Q}$ can be written as $r = \frac{m}{n}$, where $m \in \mathbb{Z}$ and $n \in \mathbb{N}$. For $x > 0$, we now define

$$y = x^r = x^{\frac{m}{n}}$$

From this, you obtain (using, among other things, rule (5)):

$$y^n = \left(x^{\frac{m}{n}}\right)^n = x^{\frac{m}{n} \cdot n} = x^m$$

Finally, by using the concept of radicals, we obtain

$$y^n = x^m \qquad \Longleftrightarrow \qquad y = \sqrt[n]{x^m}$$

Note that because $x > 0$, it follows that $y > 0$ as well. We are now ready to state ***the extended concept of exponentiation***.

The extended concept of exponentiation

Let $m \in \mathbb{Z}$ and let $n \in \mathbb{N}$ such that $\frac{m}{n} \in \mathbb{Q}$. Then the following applies:

$$x^{\frac{m}{n}} = \sqrt[n]{x^m} \qquad x > 0$$

And more specifically, the following holds true

$$x^{\frac{1}{n}} = \sqrt[n]{x} \qquad x > 0$$

The denominator of a rational exponent corresponds to the index of the radical, while the numerator remains as the exponent of the base. Conversely, the index of a radical can be transformed into the denominator of an exponent in an equivalent exponential expression. This property allows us to convert any radical expression into an exponential form, providing a powerful tool for simplification.

Example 1.18

$$\sqrt[5]{x^3} = x^{\frac{3}{5}} \quad \text{vs.} \quad \sqrt[3]{x^5} = x^{\frac{5}{3}}$$

$$\frac{1}{\sqrt[7]{x^3}} = x^{-\frac{3}{7}}$$

$$\frac{1}{\sqrt[3]{x^2}} = \left(x^2\right)^{-\frac{2}{3}}$$

This property can also be reversed: any rational exponent can be rewritten as a radical expression by using the denominator as the radical's index. The ability to interchange between exponential and radical forms enables us to evaluate expressions that were previously difficult to handle by converting them into radicals.

Example 1.19

$$27^{-\frac{4}{3}} = \frac{1}{\sqrt[3]{27^4}} = \frac{1}{\left(\sqrt[3]{27}\right)^4} = \frac{1}{3^4} = \frac{1}{81}$$

One of the greatest advantages of converting a radical expression into an exponential form is that it allows us to apply all the properties of exponents to simplify the expression. The following examples illustrate how various properties can be utilised to simplify expressions with rational exponents.

Example 1.20

$$a^{\frac{2}{3}}b^{\frac{1}{2}}a^{\frac{1}{6}}b^{\frac{1}{5}} = a^{\frac{2}{3}+\frac{1}{6}}b^{\frac{1}{2}+\frac{1}{5}} = a^{\frac{4}{6}+\frac{1}{6}}b^{\frac{5}{10}+\frac{2}{10}} = a^{\frac{5}{6}}b^{\frac{7}{10}}$$

$$\left(x^{\frac{1}{3}}x^{\frac{2}{5}}\right)^{\frac{3}{4}} = x^{\frac{1}{3}\times\frac{3}{4}}x^{\frac{2}{5}\times\frac{3}{4}} = x^{\frac{3}{12}}x^{\frac{6}{20}} = x^{\frac{1}{4}}x^{\frac{3}{10}} = x^{\frac{11}{20}}$$

$$\frac{x^{\frac{4}{2}}x^{\frac{4}{6}}x^{\frac{1}{2}}x^{\frac{5}{6}}}{x^{\frac{7}{2}}x^0} = 2x^{\frac{4}{2}+\frac{1}{2}}x^{\frac{4}{6}+\frac{5}{6}}x^{\frac{7}{2}} = 2x^{\frac{5}{2}}x^{\frac{9}{6}}x^{\frac{7}{2}} = 2x^{-1}x^{\frac{3}{2}} = 2x^{\frac{1}{2}}$$

$$\left(25x^{\frac{1}{3}}x^{\frac{2}{5}}\right)^{-\frac{1}{2}} = \left(25x^{\frac{5}{15}}x^{\frac{4}{10}}\right)^{-\frac{1}{2}} = \left(25x^{-\frac{7}{15}}x^{\frac{19}{10}}\right)^{-\frac{1}{2}} = \left(\frac{9}{25x^{-\frac{7}{15}}x^{\frac{19}{10}}}\right)^{\frac{1}{2}} = \frac{9}{2}\cdot 25x^{-\frac{7}{30}}x^{\frac{19}{20}} = \frac{3x^{\frac{7}{30}}}{5x^{\frac{19}{20}}}$$

It is important to remember that when simplifying expressions with rational exponents, we are applying the same exponent rules that are used for integer exponents. The only difference is that we must also adhere to the rules for fractions.

1.4 Using Formulae and Substitution

In the study of engineering, physical quantities are often related to each other through formulas. These formulas consist of variables and constants that represent the physical quantities in question. To evaluate a formula, one must substitute numerical values for the variables.

For example, Ohm's law provides a formula that relates the voltage, v, across a resistor with a resistance value R, to the current i flowing through it. The formula is given by

$$v = iR$$

This formula allows us to calculate the voltage v if the values for i and R are known. For instance, if $i = 13\,\text{A}$ and $R = 5\,\Omega$, then

$$v = iR = (13)(5) = 65$$

Thus, the voltage is 65 V.

This example highlights the importance of paying close attention to the units of any physical quantities involved. A formula is only valid if a consistent set of units is used.

Example 1.21 Inserting into formulae
The kinetic energy K of an object with mass M moving at speed v can be calculated using the formula:

$$K = \frac{1}{2}Mv^2$$

Calculate the kinetic energy of an object with a mass of 5 kg moving at a speed of $2\,\text{m}\,\text{s}^{-1}$.

Solution:

$$K = \frac{1}{2}Mv^2 = \frac{1}{2}(5)(2^2) = 10$$

◀

In the SI system, the unit of energy is the joule, so the kinetic energy of the object is 10 joules.

Example 1.22 Inserting into formulae
The area A of a circle with radius r can be calculated using the formula $A = \pi r^2$.

Alternatively, if the diameter d of the circle is known, the equivalent formula can be used:

$$A = \frac{\pi d^2}{4}$$

Calculate the area of a circle with a diameter of 0.1 m. The value of π is pre-programmed in your calculator.

Solution:

$$A = \frac{\pi(0.1)^2}{4} = 0.00785\,\text{m}^2$$

◀

Example 1.23 Inserting into formulae
The volume V of a circular cylinder is equal to its cross-sectional area A multiplied by its length h.

Calculate the volume of a cylinder with a diameter of 0.1 m and a length of 0.3 m.

Solution:

$$V = Ah = \frac{\pi(0.1)^2}{4} \times 0.3 = 0.00236$$

The volume is $0.00236\,\text{m}^3$. ◀

1.5 Rearranging Formulae

In the formula for the area of a circle, $A = \pi r^2$, the variable A is referred to as the subject of the formula. A variable is considered the subject if it appears by itself on one side of the equation, usually on the left-hand side, and nowhere else in the formula. If we are asked to transpose the formula for r, or solve for r, we must rearrange the equation so that r becomes the subject. When transposing a formula, any operation performed on one side must also be applied to the other side. There are five key rules to follow during this process.

Rules for rearranging formulae

The following operations can be performed on both sides of the formula:

- Add the same quantity to both sides
- Subtract the same quantity from both sides
- Multiply both sides by the same quantity - remember to multiply all terms
- Divide both sides by the same quantity - remember to divide all terms
- Apply a function to both sides, such as squaring or finding the reciprocal

Example 1.24 Transpose the formula $p = 5t - 17$ to make t the subject.

Solution: To isolate t on the left-hand side, proceed in steps using the five rules. First, add 17 to both sides of the equation $p = 5t - 17$:

$$p + 17 = 5t - 17 + 17$$

Simplifying, we get:

$$p + 17 = 5t$$

Next, divide both sides by 5 to isolate t:

$$\frac{p + 17}{5} = t$$

Thus, the formula for t is:

$$t = \frac{p + 17}{5}$$

◀

Example 1.25 Transpose the formula $\sqrt{2q} = p$ to solve for q.

Solution: First, square both sides to eliminate the square root around $2q$. Note that $(\sqrt{2q})^2 = 2q$. This gives:

$$2q = p^2$$

Next, divide both sides by 2 to solve for q:

$$q = \frac{p^2}{2}$$

◀

Problem 1.1 Transpose the formula $v = \sqrt{t^2 + w}$ to solve for w. To isolate w, follow these steps:

a. First, square both sides to eliminate the square root around $t^2 + w$:
$$v^2 = t^2 + w$$
b. Next, subtract t^2 from both sides to isolate w:
$$v^2 - t^2 = w$$
c. Finally, write down the formula for w:
$$w = v^2 - t^2$$

Example 1.26 Transpose the formula $x = \frac{1}{y}$ to solve for y.

Solution: To isolate y, notice that y appears in the denominator. Multiplying both sides by y removes the fraction:

$$yx = y \times \frac{1}{y}$$

This simplifies to:

$$yx = 1$$

Finally, divide both sides by x to solve for y:

$$y = \frac{1}{x}$$

Alternatively, you can simply invert both sides directly to obtain:

$$y = \frac{1}{x}$$

◀

Example 1.27 Make R the subject of the formula:

$$\frac{2}{R} = \frac{3}{x+y}$$

Solution: Since R appears in a fraction, invert both sides:

$$\frac{R}{2} = \frac{x+y}{3}$$

Multiplying both sides by 2 yields: $R = \dfrac{2(x+y)}{3}$ ◀

Example 1.28 Make R the subject of the formula:

$$\frac{1}{R} = \frac{1}{R_1} + \frac{1}{R_2}$$

Solution: The two terms on the right-hand side can be combined:

$$\frac{1}{R_1} + \frac{1}{R_2} = \frac{R_2 + R_1}{R_1 R_2}$$

The formula then becomes:

$$\frac{1}{R} = \frac{R_2 + R_1}{R_1 R_2}$$

Finally, inverting both sides gives:

$$R = \frac{R_1 R_2}{R_2 + R_1}$$

1.6 Functions

In mathematics, a function assigns each element of one set to a specific element of another set (which may be the same set). For example, consider a Mathematics for Software Engineering class where each student is assigned a grade from the set $\{12, 10, 7, 4, 02\}$. Suppose the grades are as follows:

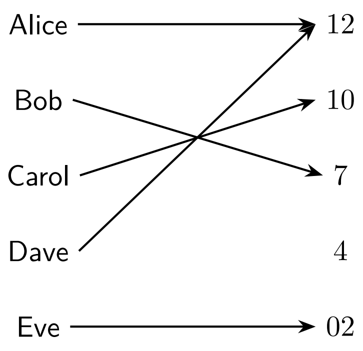


Figure 1.1: Example of a function mapping names to numbers.

This assignment of grades, illustrated in figure 1.1, exemplifies a function.

Functions play a crucial role in mathematics and computer science. They define discrete structures such as sequences and strings and are used to analyse the time complexity of algorithms. Many computer programs are designed to compute values of functions. Recursive functions, defined in terms of themselves, are especially significant in computer science. This section provides an overview of the fundamental concepts of functions needed in the mathematics for software engineering.

Definition 1.1

Let A and B be nonempty sets. A function f from A to B is an assignment of exactly one element of B to each element of A. We write $f(a) = b$ if b is the unique element of B assigned by the function f to the element a of A. If f is a function from A to B, we write $f : A \to B$. ♣

Remark: Functions are sometimes also called mappings or transformations.

Functions can be specified in various ways. Sometimes, we explicitly state the assignments, as shown in Figure 1.1. Often, a formula such as $f(x) = x + 1$ is used to define a function. In other cases, a computer program may specify the function.

Definition 1.2

If f is a function from A to B, we say that A is the **domain** of f and B is the **co-domain** of f. If $f(a) = b$, we say that b is the **image** of a and a is a **preimage** of b. The **range**, or image, of f is the set of all images of elements of A. Also, if f is a function from A to B, we say that f **maps** A to B. ♣

When defining a function, we specify its domain, co-domain, and the mapping of elements from the domain to the co-domain. Two functions are equal if they have the same domain, the same co-domain, and map each element of their domain to the same element in the co-domain.

It's important to note that altering the domain or co-domain results in a different function. Similarly, changing the mapping of elements also produces a different function.

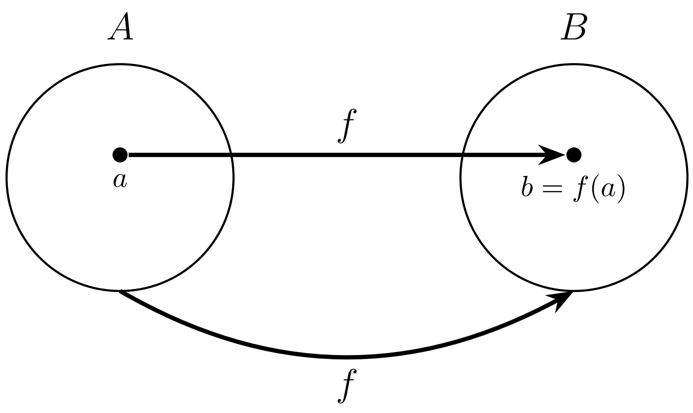


Figure 1.2: A function f mapping an element a from set A to an element $b = f(a)$ in set B.

The following examples illustrate various functions. In each example, we describe the domain, co-domain, range, and the assignment of values to the elements of the domain.

Example 1.29 What are the domain, co-domain, and range of the function that assigns grades to students described in the first paragraph of the introduction of this section?

Solution: Let G be the function that assigns a grade to a student in our Software engineering mathematics class. Note that $G(\text{Alice}) = 12$, for instance. The domain of G is the set {Alice, Bob, Carol, David, Eve }, and the co-domain is the set $\{12, 10, 7, 4, 02\}$. The range of G is the set $\{12, 10, 7, 02\}$, because each grade except 4 is assigned to some student. ◀

Example 1.30 Let f be the function that assigns the last two bits of a bit string of length 2 or greater to that string. For example, $f(11010) = 10$. Then, the domain of f is the set of all bit strings of length 2 or greater, and both the co-domain and range are the set $\{00, 01, 10, 11\}$.

Example 1.31 Let $f : \mathbb{Z} \to \mathbb{Z}$ assign the square of an integer to this integer. Then, $f(x) = x^2$, where the domain of f is the set of all integers, the co-domain of f is the set of all integers, and the range of f is the set of all integers that are perfect squares, namely, $\{0, 1, 4, 9, \ldots\}$.

One-to-One and Onto Functions

In mathematics, functions are a fundamental concept used to describe the relationship between two sets. However, not all functions behave the same way. To understand these differences, we introduce the concepts of one-to-one (injective) and onto (surjective) functions.

Some functions never assign the same value to two different domain elements. These functions are said to be **one-to-one**.

Definition 1.3 (One-to-One functions (Injective))

A function $f : A \to B$ is called **one-to-one** (or **injective**) if different elements in A map to different elements in B. In other words, if $f(a_1) = f(a_2)$, then $a_1 = a_2$. This property ensures that no two distinct elements in A are mapped to the same element in B. ♣

Graphically, a function is one-to-one if no horizontal line intersects the graph of the function at more than one point.

Definition 1.4 (Onto Functions (Surjective))

A function $f : A \to B$ is called **onto** (or **surjective**) if every element in B is the image of at least one element in A. In other words, for every $b \in B$, there exists at least one $a \in A$ such that $f(a) = b$. This property ensures that the function "covers" the entire set B. ♣

Inverse Functions

Now, consider a function $f : A \to B$ that is both one-to-one and onto. Because f is onto, every element of B is the image of some element in A. Furthermore, because f is one-to-one, every element of B is the image of a unique element of A. This unique correspondence allows us to define a new function from B to A that "reverses" the mapping given by f.

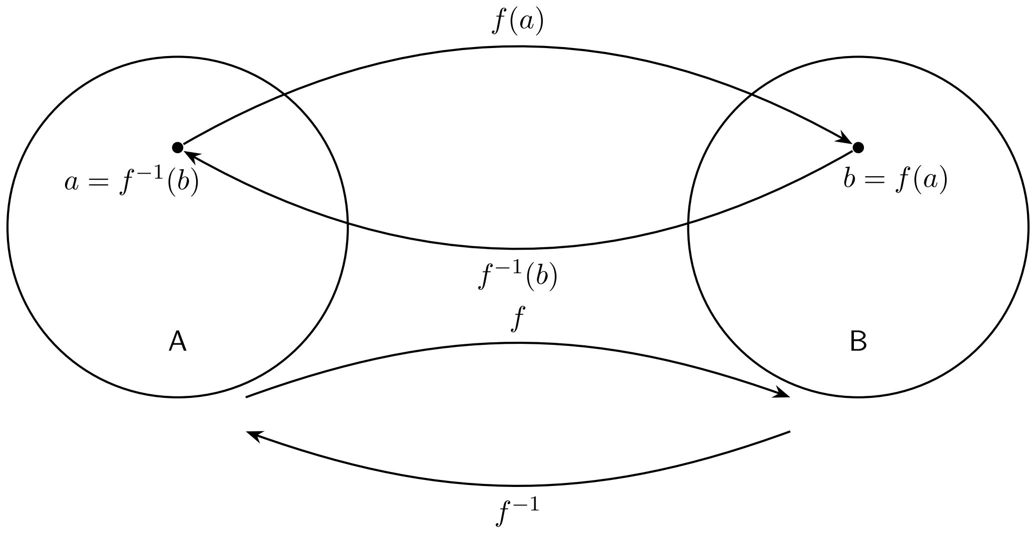


Figure 1.3: The function f^{-1} is the inverse of function f.

This new function is called the **inverse function** of f, denoted by $f^{-1} : B \to A$. The inverse function f^{-1} satisfies the following properties:

$$f(f^{-1}(b)) = b \quad \text{for every } b \in B.$$

$$f^{-1}(f(a)) = a \quad \text{for every } a \in A.$$

We can summarise these considerations in the following definition

Definition 1.5 (Inverse Functions)

Let f be a one-to-one correspondence from the set A to the set B. The inverse function of f is the function that assigns to an element b belonging to B the unique element a in A such that $f(a) = b$. The inverse function of f is denoted by f^{-1}. Hence, $f^{-1}(b) = a$ when $f(a) = b$. ♣

These properties show that f^{-1} effectively undoes the work of f, mapping each element of B back to the corresponding element in A.

Example 1.32

Let $f : \mathbb{R} \to \mathbb{R}$ be defined by $f(x) = 2x + 3$. We can check that f is both one-to-one and onto:

- **One-to-One**: If $f(x_1) = f(x_2)$, then $2x_1 + 3 = 2x_2 + 3$. Subtracting 3 from both sides gives $2x_1 = 2x_2$, and dividing by 2 yields $x_1 = x_2$. Thus, f is one-to-one.
- **Onto**: Given any $y \in \mathbb{R}$, we can solve $y = 2x + 3$ for x to find $x = \frac{y-3}{2}$. Since this x exists for every y, f is onto.

Since f is both one-to-one and onto, it has an inverse function f^{-1} defined by

$$f^{-1}(y) = \frac{y-3}{2}.$$

Composite Functions

In mathematics, functions can be combined to form new functions. One important way of combining functions is through the composition of functions. The composite of two functions is essentially applying one function to the results of another.

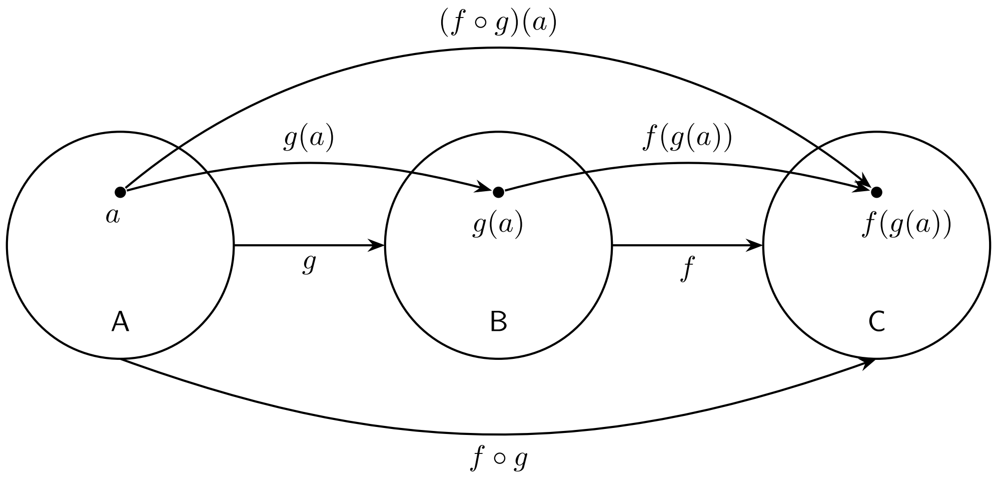


Figure 1.4: The composition of f and g.

Definition 1.6

Let $f : B \to C$ and $g : A \to B$ be two functions. The **composite function** of f and g, denoted by $f \circ g$, is a function from A to C defined by

$$(f \circ g)(x) = f(g(x)),$$

for every $x \in A$. ♣

In other words, the composite function $f \circ g$ means that you first apply the function g to the input x, and then apply the function f to the result of $g(x)$. In figure 1.4 the composition of functions is shown.

Example 1.33

Consider the functions $f(x) = 2x + 3$ and $g(x) = x^2$. The composite function $f \circ g$ is given by:

$$(f \circ g)(x) = f(g(x)) = f(x^2) = 2x^2 + 3.$$

Here, the function $g(x)$ squares the input x, and then the function $f(x)$ multiplies the result by 2 and adds 3.

Now, let's reverse the composition and compute $g \circ f$:

$$(g \circ f)(x) = g(f(x)) = g(2x + 3) = (2x + 3)^2.$$

Notice that $f \circ g$ and $g \circ f$ are generally different functions, illustrating that the composition of functions is not commutative.

Example 1.34 Let g be the function from the set $\{a, b, c\}$ to itself such that $g(a) = b, g(b) = c$, and $g(c) = a$. Let f be the function from the set $\{a, b, c\}$ to the set $\{1, 2, 3\}$ such that $f(a) = 3, f(b) = 2$, and $f(c) = 1$. What is the composition of f and g, and what is the composition of g and f ?

Solution: The composition $f \circ g$ is defined by $(f \circ g)(a) = f(g(a)) = f(b) = 2$, $(f \circ g)(b) = f(g(b)) = f(c) = 1$, and $(f \circ g)(c) = f(g(c)) = f(a) = 3$.

Note that $g \circ f$ is not defined, because the range of f is not a subset of the domain of g. ◀

Example 1.35 label: Let f and g be the functions from the set of integers to the set of integers defined by $f(x) = 2x + 3$ and $g(x) = 3x + 2$. What is the composition of f and g ? What is the composition of g and f?

Solution: Both the compositions $f \circ g$ and $g \circ f$ are defined. Moreover,

$$(f \circ g)(x) = f(g(x)) = f(3x + 2) = 2(3x + 2) + 3 = 6x + 7$$

and

$$(g \circ f)(x) = g(f(x)) = g(2x + 3) = 3(2x + 3) + 2 = 6x + 11.$$

◀

Remark: Even though $f \circ g$ and $g \circ f$ are defined for the functions f and g in example 35, $f \circ g$ and $g \circ f$ are not equal. In other words, the commutative law does not hold for the composition of functions.

When the composition of a function and its inverse is formed, in either order, an identity function is obtained. To see this, suppose that f is a one-to-one correspondence from the set A to the set B. Then the inverse function f^{-1} exists and is a one-to-one correspondence from B to A. The inverse function reverses the correspondence of the original function, so $f^{-1}(b) = a$ when $f(a) = b$, and $f(a) = b$ when $f^{-1}(b) = a$.

Hence,

$$\left(f^{-1} \circ f\right)(a) = f^{-1}(f(a)) = f^{-1}(b) = a,$$

and

$$\left(f \circ f^{-1}\right)(b) = f\left(f^{-1}(b)\right) = f(a) = b.$$

Consequently, $f^{-1} \circ f = I_A$ and $f \circ f^{-1} = I_B$, where I_A and I_B are the identity functions on the sets A and B, respectively. That is, $\left(f^{-1}\right)^{-1} = f$.

Example 1.36 If $f : \mathbb{R} \to \mathbb{R}$ is defined as $f(x) = 2x + 3$, then $f^{-1}(x) = \frac{x-3}{2}$. The composition $f \circ f^{-1}$ would be the identity function $I_{\mathbb{R}}$ on the real numbers, meaning $f\left(f^{-1}(x)\right) = x$ for all $x \in \mathbb{R}$.

1.7 Graphical Identification of Function Types

Understanding the behaviour of different types of functions is fundamental in mathematics. Functions can be classified based on their graphical patterns, which provide valuable insights into their characteristics. In this section, we will explore various types of functions, including linear, quadratic, exponential, and more. By examining their graphs, we can identify key features such as intercepts, slopes, curvature, and asymptotic behaviour, enabling us to distinguish between these different types of functions effectively.

Linear Functions

The general equation for a linear function is given by

$$y = ax + b \quad \text{(often written as } y = mx + b\text{)},$$

where a (or m) represents the slope and b is the y-intercept. The domain of this function is all real numbers. This equation is in slope-intercept form because a (or m) gives the slope and b gives the y-intercept. If $a = 0$, the function simplifies to $y = b$, which is a constant function.

The parent function for a linear equation is

$$y = x.$$

The transformed function can be written in the point-slope form as

$$y = y_1 + a(x - x_1),$$

where the graph contains the point (x_1, y_1) and has slope a. In this form:

- a is the vertical dilation (slope),
- y_1 represents the vertical translation,
- x_1 represents the horizontal translation.

This point-slope form can also be written as

$$y - y_1 = a(x - x_1),$$

where the coordinates of the fixed point (x_1, y_1) appear with a negative sign. The form $y = y_1 + a(x - x_1)$ expresses y explicitly in terms of x, making it easier to enter into a graphing calculator.

The graph of a linear function is a straight line. The parent function $y = x$ is shown on the left in figure 1.5, the slope-intercept form in the middle, and the point-slope form on the right.

For the slope-intercept form: "Start at b on the y-axis, move x units horizontally, and rise ax units vertically." For the point-slope form: "Start at (x_1, y_1), move $(x - x_1)$ units horizontally, and rise $a(x - x_1)$ units vertically."

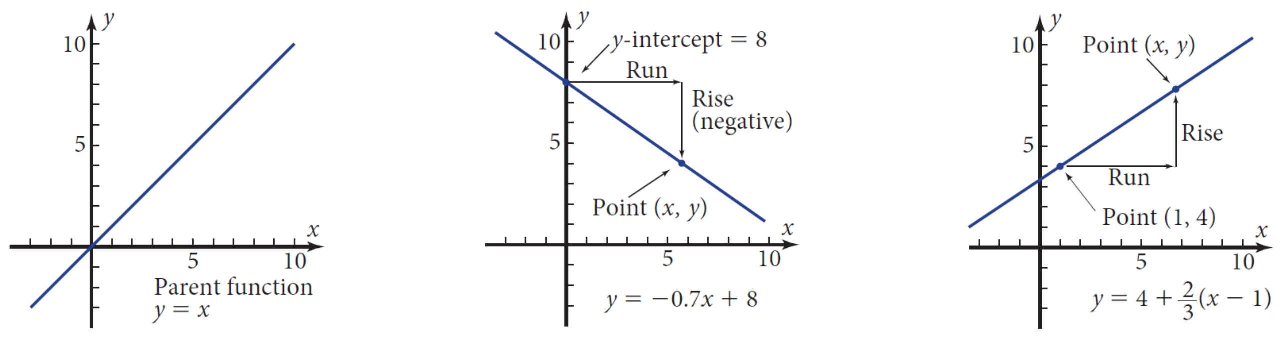


Figure 1.5: Linear functions

Quadratic Functions

The general equation for a quadratic function is given by

$$y = ax^2 + bx + c,$$

where $a \neq 0$, and a, b, and c are constants. The domain of this function is all real numbers.

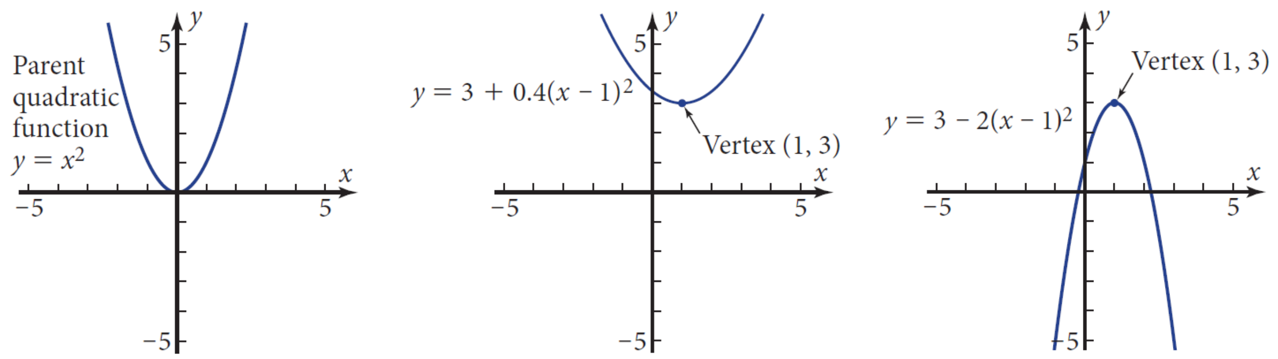


Figure 1.6: Quadratic functions

The parent function for a quadratic equation is

$$y = x^2,$$

where the vertex of the parabola is at the origin $(0, 0)$.

The transformed function can be written in vertex form as

$$y = k + a(x - h)^2,$$

where the vertex of the parabola is located at (h, k). In this form:

- k represents the vertical translation,
- h represents the horizontal translation,
- a represents the vertical dilation.

Vertex form can also be written as

$$y - k = a(x - h)^2,$$

but expressing y explicitly in terms of x makes the equation easier to enter into a graphing calculator.

The graph of a quadratic function is a parabola (from the Greek word for "along the path of a ball"). The parabola is concave up if $a > 0$ and concave down if $a < 0$. This behaviour is illustrated in figure 1.6.

Power Functions

The general equation for a power function is given by

$$y = ax^b,$$

where a and b are nonzero constants. The domain of the function depends on the value of b:

- If $b > 0$, the domain is all real numbers.
- If $b < 0$, the domain excludes $x = 0$ to avoid division by zero.
- If b is not an integer, the domain usually excludes negative numbers to avoid taking roots of negative numbers.

In most applications, the domain is restricted to non-negative numbers.

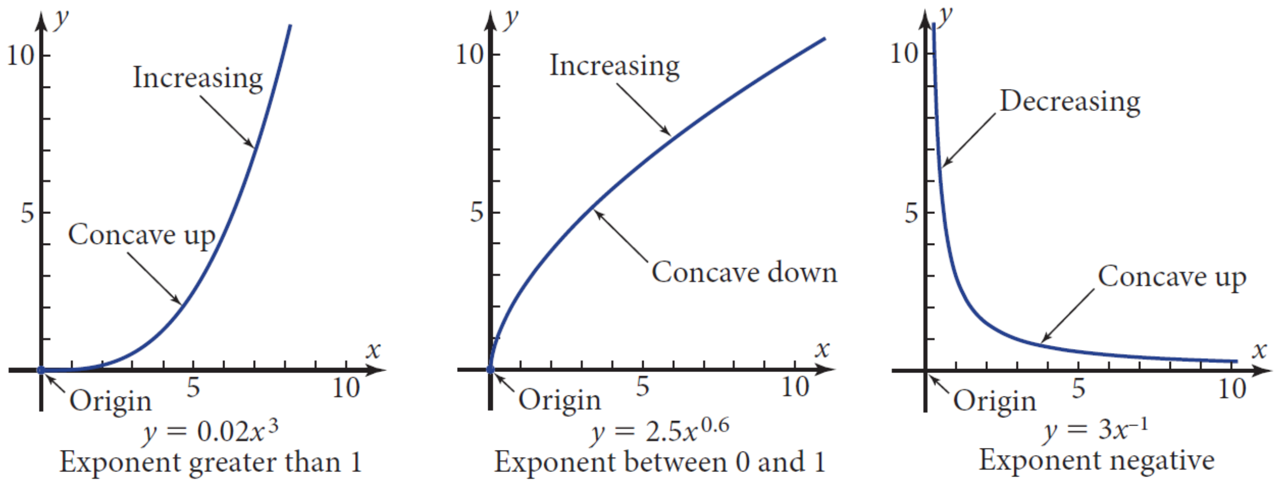


Figure 1.7: Power functions

The parent function for a power function is

$$y = x^b.$$

For the general power function $y = ax^b$:

- If $b > 0$, then y varies directly with the bth power of x, meaning y is directly proportional to the bth power of x.
- If $b < 0$, then y varies inversely with the bth power of x, meaning y is inversely proportional to the bth power of x.

The dilation factor a serves as the proportionality constant.

The translated form of a power function is

$$y = d + a(x - c)^b,$$

where c and d are the horizontal and vertical translations, respectively. This can be compared with the translated forms of linear and quadratic functions:

$$y = y_1 + a(x - x_1) \qquad \text{(linear function)},$$
$$y = k + a(x - h)^2 \qquad \text{(quadratic function)}.$$

Unless otherwise stated, "power function" will imply the untranslated form, $y = ax^b$.

Figure 1.7 shows the graphs of power functions for different values of b. In all cases, $a > 0$. The shape and concavity of the graph depend on the value of b:

- If $b > 0$, the graph contains the origin.
- If $b < 0$, the graph has the axes as asymptotes.
- The function is increasing if $b > 0$ and decreasing if $b < 0$.
- The graph is concave up if $b > 1$ or $b < 0$, and concave down if $0 < b < 1$.

The concavity of the graph describes the rate at which y increases. For $b > 0$, concave up indicates that y is increasing at an increasing rate, while concave down indicates that y is increasing at a decreasing rate.

Exponential Functions

The general equation for an exponential function is given by

$$y = ab^x,$$

where a and b are constants, $a \neq 0$, $b > 0$, and $b \neq 1$. The domain of this function is all real numbers.

The parent function for an exponential equation is

$$y = b^x,$$

where the asymptote is the x-axis.

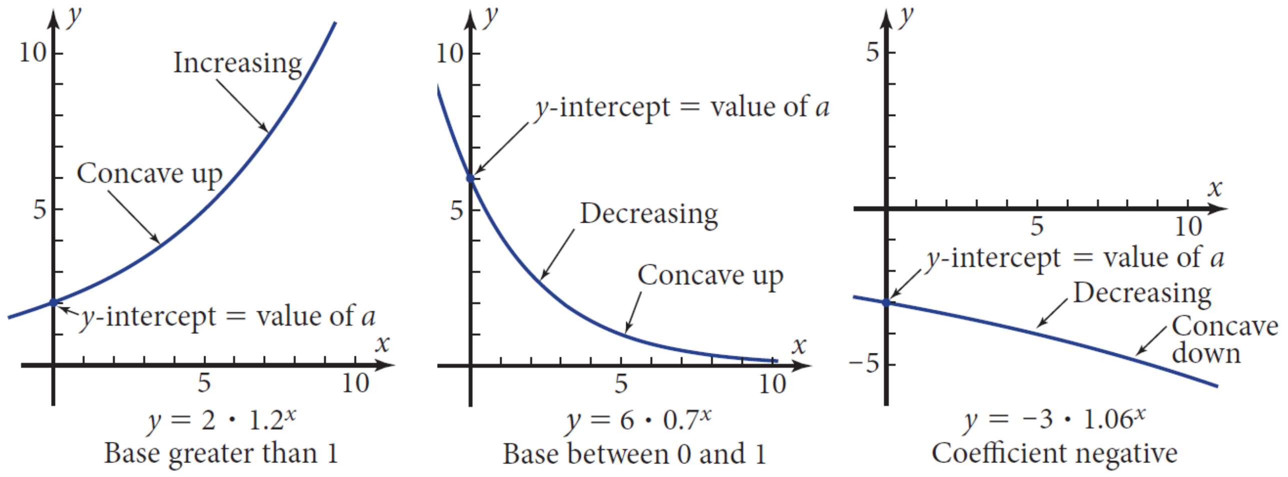


Figure 1.8: Exponential functions

In the equation $y = ab^x$, we say that "y varies exponentially with x." This means that y changes by a constant factor b for each unit increase in x.

The translated form of the exponential function is

$$y = ab^x + c,$$

where the asymptote is the line $y = c$. Unless otherwise stated, "exponential function" will refer to the untranslated form $y = ab^x$.

Figure 1.8 illustrates exponential functions for different values of a and b. The key properties of the graph are as follows:

- The constant a is the y-intercept of the graph.
- The function is increasing if $b > 1$ and decreasing if $0 < b < 1$, provided $a > 0$.
- If $a < 0$, the function's behavior is reversed: it is decreasing if $b > 1$ and increasing if $0 < b < 1$.
- The graph is concave up if $a > 0$ and concave down if $a < 0$.

Mathematicians often use one of two particular constants as the base for an exponential function: either 10, which is the base of the decimal system, or the naturally occurring number e, which approximately equals 2.71828. These bases are significant in various mathematical applications.

Definition 1.7 (Special Exponential Functions)

$y = a \cdot 10^{bx}$	base-10 exponential function
$y = a \cdot e^{bx}$	natural (base-e) exponential function,

where a and b are constants and the domain is all real numbers. ♣

To generalise the exponential function, the variable in the exponent is often multiplied by a constant. The (untranslated) general forms of these exponential functions are given below:

$$y = a \cdot 10^{bx} \quad \text{and} \quad y = a \cdot e^{bx}$$

These functions can be further generalised by incorporating translations in both the x- and y-directions. The translated forms are:

$$y = a \cdot 10^{b(x-c)} + d \quad \text{and} \quad y = a \cdot e^{b(x-c)} + d$$

The base-e exponential function, in particular, has a significant advantage when studying calculus, as the rate of change of e^x is equal to e^x itself.

1.8 Logarithms

Any positive number can be written as a power of 10. For instance,

$$\begin{aligned} 3 &= 10^{0.477\ldots} \\ 5 &= 10^{0.6989\ldots} \\ 15 &= 10^{1.1760\ldots} \end{aligned}$$

The exponents $0.4771\ldots$, $0.6989\ldots$, and $1.1760\ldots$ are called the base-10 logarithms of 3, 5, and 15, respectively:

$$\begin{aligned} \log 3 &= 0.4771\ldots \\ \log 5 &= 0.6989\ldots \\ \log 15 &= 1.1760\ldots \end{aligned}$$

To better understand the meaning of logarithms, press `LOG` 3 on your calculator. You will get:

$$\log 3 = 0.477121254\ldots$$

Then, without rounding, raise 10 to this power. You will obtain:

$$10^{0.477121254\ldots} = 3$$

The powers of 10 have the normal properties of exponentiation. For instance,

$$\begin{aligned}15 = (3)(5) &= \left(10^{0.4771\ldots}\right)\left(10^{0.6999\ldots}\right)\\ &= 10^{0.4771\ldots+0.6599\ldots}\\ &= 10^{1.1760\ldots}\end{aligned}$$

This means $10^{0.4771\ldots+0.6599\ldots} = 10^{1.1760\ldots}$. Here, you add the exponents while keeping the same base. You can verify with your calculator that $10^{1.1760\ldots}$ indeed equals 15.

From this example, you can infer that logarithms have the same properties as exponents. This is expected because logarithms *are* exponents. For instance,

$$\log(3\cdot 5) = \log 3 + \log 5 \quad \textit{The logarithm of a product equals the sum of the logarithms of the factors.}$$

From the values given earlier, you can also show that:

$$\log\frac{15}{3} = \log 15 - \log 3 \quad \textit{The logarithm of a quotient.}$$

This property is reasonable because you divide powers of equal bases by subtracting the exponents:

$$\frac{15}{3} = \frac{10^{1.1760\ldots}}{10^{0.477\ldots}} = 10^{1.1760\ldots-0.4771\ldots} = 10^{0.6989\ldots} = 5$$

Since a power can be written as a product, you can find the logarithm of a power as follows:

$$\begin{aligned}\log 34 = \log(3\cdot 3\cdot 3\cdot 3) &= \log 3 + \log 3 + \log 3 + \log 3\\ &= 4\log 3 \quad \textit{Combine like terms.}\end{aligned}$$

The logarithm of a power equals the exponent of that power times the logarithm of the base. To verify this result, observe that $3^4 = 81$. Press $4 \times$ `LOG` 3 on your calculator, and you'll find it equals $1.9084\ldots$.

Definition 1.8 (Base-10 Logarithms)

$$\log x = y \iff 10^y = x$$

Verbally: $\log x$ is the exponent in the power of 10 that gives x ♣

The term logarithm comes from the Greek words *logos*, meaning "ratio," and *arithmos*, meaning "number." Before the invention of calculators, base-10 logarithms were calculated approximately using infinite series and recorded in tables. Products involving many factors, such as

$$(357)(4.367)(22.4)(3.142)$$

could be calculated by adding their logarithms (exponents) rather than tediously multiplying several pairs of numbers. This method was invented by Englishman Henry Briggs (1561–1630) and Scotsman John Napier (1550–1616). The name logarithm, thus, reflects this "logical way to do arithmetic".

Properties of base-10 logarithms

- Log of a Product:

  $\log xy = \log x + \log y$

  Verbally: The $\log$ of a product equals the sum of the logs of the factors.

- Log of a Quotient:

  $\log \frac{x}{y} = \log x - \log y$

  Verbally: The $\log$ of a quotient equals the log of the numerator minus the $\log$ of the denominator.

- Log of a Power:

  $\log x^y = y \log x$

  Verbally: The $\log$ of a power equals the exponent times the log of the base.

Example 1.37 Find x if $\log_{10} 10^{3.721} = x$

Solution: By definition, the logarithm is the exponent of 10. So $x = 3.721$. ◀

Example 1.38 Find x if $0.258 = 10^x$

Solution: By definition, x, the exponent of 10 , is the logarithm of 0.258 .

$$x = \log_{10} 0.258 = -0.5883\ldots$$

◀

The most important thing to remember about logarithms is this

A logarithm is an exponent.

Logarithms with Any Base: The Change-of-Base Property

If $x = 10^y$, then y is the base-10 logarithm of x. Similarly, if $x = 2^y$, then y is the base-2 logarithm of x. The only difference between these logarithms is the number that serves as the base. To distinguish among logarithms with different bases, the base is written as a subscript after the abbreviation "log." For instance:

$$3 = \log_2 8 \Leftrightarrow 2^3 = 8,$$
$$4 = \log_3 81 \Leftrightarrow 3^4 = 81,$$
$$2 = \log_{10} 100 \Leftrightarrow 10^2 = 100.$$

The symbol $\log_2 8$ is pronounced "log to the base 2 of 8." The symbol $\log_{10} 100$ is, of course, equivalent to $\log 100$, as defined in the previous section. Note that in all cases, a logarithm represents an exponent.

Definition 1.9 (Logarithm with Any Base)

Algebraically:

$\log_b x = y$ if and only if $b^y = x$, where $b > 0, b \neq 1$, and $x > 0$

Verbally:

$\log_b x = y$ means that y is the exponent of b that gives x as the answer. ♣

The way you pronounce the symbol for logarithm gives you a way to remember the definition. The next two examples show you how to do this.

Example 1.39 Write $\log_5 c = a$ in exponential form.

Solution:

Think this:

- "$\log_5 \ldots$" is read as "log base 5 . . .," meaning 5 is the base.
- A logarithm is an exponent. Since the $\log$ equals a, a must be the exponent.
- The "answer" obtained from 5^a is the argument of the logarithm, denoted as c.

Write only this:

$$5^a = c$$

◀

Example 1.40 Write $z^4 = m$ in logarithmic form.

Solution: $\log_z m = 4$ ◀

Two bases of logarithms are used frequently enough to have their own key on most calculators. One is the base-10 logarithm, also known as the common logarithm, as discussed in the previous section. The other is the base-e logarithm, known as the natural logarithm, where $e = 2.71828\ldots$, a naturally occurring number (like π) that will be advantageous in your future mathematical studies.

The symbol $\ln x$ (pronounced "el en of x") is used for natural logarithms, and is defined as:

$$\ln x = \log_e x$$

Definition 1.10 (Common Logarithm and Natural Logarithm)

Common: The symbol $\log x$ means $\log_{10} x$.

Natural: The symbol $\ln x$ means $\log_e x$, where e is a constant equal to $2.71828182845\ldots$ ♣

Example 1.41 Find $\log_5 17$. Check your answer by an appropriate numerical method.

Solution: Let $x = \log_5 17$.

$$5^x = 17$$
$$\log_{10} 5^x = \log_{10} 17$$
$$x \log_{10} 5 = \log_{10} 17$$
$$x = \frac{\log_{10} 17}{\log_{10} 5} = 1.7603\ldots$$
$$\log_5 17 = 1.7603\ldots$$
$$5^{1.7603\ldots} = 17$$

◀

In this example, note that the base-5 logarithm of a number is directly proportional to the base-10 logarithm of that number. The conclusion of the example can be expressed as follows:

$$\log_5 17 = \frac{1}{\log_{10} 5} \cdot \log_{10} 17 = 1.4306\ldots \log_{10} 17$$

To find the base-5 logarithm of any number, simply multiply its base-10 logarithm by $1.4306\ldots$ (that is, divide by $\log_{10} 5$).

This proportional relationship is known as the change-of-base property. From the results of Example 3, you can write:

$$\log_5 17 = \frac{\log_{10} 17}{\log_{10} 5}$$

Notice that the logarithm with the desired base is isolated on the left side of the equation, while the two logarithms on the right side share the same base—typically one that is available on your calculator. The box below illustrates this property for bases a and b with argument x:

The Change-of-Base Property of Logarithms

$$\log_a x = \frac{\log_b x}{\log_b a} \quad \text{or} \quad \log_a x = \frac{1}{\log_b a}(\log_b x)$$

Example 1.42 Find $\ln 29$ using the change-of-base property with base-10 logarithms. Check your answer directly by pressing $\ln 29$ on your calculator.

Solution:

$$\ln 29 = \frac{\log 29}{\log e} = \frac{1.4623\ldots}{0.4342\ldots} = 3.3672\ldots$$

Directly: $\ln 29 = 3.3672\ldots,$

which agrees with the answer we got using the change-of-base property. ◀

Properties of Logarithms

The Logarithm of a Power:

$$\log_b x^y = y \log_b x$$

Verbally: The logarithm of a power equals the product of the exponent and the logarithm of the base. The Logarithm of a Product:

$$\log_b(xy) = \log_b x + \log_b y$$

Verbally: The logarithm of a product equals the sum of the logarithms of the factors. The Logarithm of a Quotient:

$$\log_b \frac{x}{y} = \log_b x - \log_b y$$

Verbally: The logarithm of a quotient equals the logarithm of the numerator minus the logarithm of the denominator.

Solving Exponential and Logarithmic Equations

Logarithms provide a way to solve an equation with a variable in the exponent or to solve an equation that already contains logarithms. We will demonstrate this through the next few examples.

Example 1.43 Solve the exponential equation $7^{3x} = 983$ algebraically, using logarithms.

Solution:

$7^{3x} = 983$

$\log 7^{3x} = \log 983$ `Take the base-10 logarithm of both sides.`

$3x \log 7 = \log 983$ `Apply the logarithm power property.`

$x = \dfrac{\log 983}{3 \log 7}$ `Divide both sides by the coefficient of x.`

$x = 1.1803\ldots$

◀

Example 1.44 Solve the equation

$$\log_2(x-1) + \log_2(x-3) = 3$$

Solution:

$\log_2(x-1) + \log_2(x-3) = 3$

$\log_2[(x-1)(x-3)] = 3$ `Apply the logarithm of a product property.`

$2^3 = (x-1)(x-3)$ `Use the definition of logarithm.`

$8 = x^2 - 4x + 3$ `Expand the product.`

$x^2 - 4x - 5 = 0$ `Reduce one side to zero. Use the symmetric property of equality.`

$(x-5)(x+1) = 0$ `Solve by factoring.`

$x = 5$ or $x = -1$

We need to be cautious here because the solutions in the final step are the solutions of the quadratic equation, and we must make sure they are also solutions of the original logarithmic equation. Check by substituting the solutions into the original equation.

If $x = 5$, then

$$\log_2(5-1) + \log_2(5-3)$$
$$= \log_2 4 + \log_2 2$$
$$= 2 + 1 = 3$$

If $x = -1$, then

$$\log_2(-1-1) + \log_2(-1-3)$$
$$= \log_2(-2) + \log_2(-4)$$

which is undefined.

◀

Example 1.45 Solve the equation

$$e^{2x} - 3e^x + 2 = 0$$

Solution:

$$e^{2x} - 3e^x + 2 = 0$$
$$(e^x)^2 - 3e^x + 2 = 0$$

We realise that this is a quadratic equation in the variable e^x. Using the quadratic formula, you get

$$e^x = \frac{+3 \pm \sqrt{9-4(2)}}{2} = \frac{3 \pm 1}{2}$$
$$e^x = 2 \text{ or } e^x = 1$$

You now have to solve these two equations.

$$\begin{array}{ll} e^x = 2 & e^x = 1 \\ x = \ln 2 = 0.6931\dots & x = 0 \end{array}$$

Check:

$$\begin{array}{ll} e^{2\ln 2} - 3e^{\ln 2} + 2 & (e^0)^2 - 3e^0 + 2 \\ = \left(e^{\ln 2}\right)^2 - 3e^{\ln 2} + 2 & = 1^2 - 3(1) + 2 = 0 \\ = 2^2 - 3(2) + 2 = 0 & \end{array}$$

Both solutions are correct. ◀

Example 1.46 Solve the logarithmic equation $\ln(x+3) + \ln(x+5) = 0$

Solution:

$$\ln(x+3) + \ln(x+5) = 0$$
$$\ln[(x+3)(x+5)] = 0$$
$$(x+3)(x+5) = e^0 = 1$$
$$x^2 + 8x + 15 = 1$$
$$x^2 + 8x + 14 = 0$$
$$x = -2.5857\dots \quad \text{or} \quad x = -5.4142\dots$$

Check:

$$x = -2.5857\dots:$$
$$\ln(-2.5857\dots + 3) + \ln(-2.5857\dots + 5)$$
$$= \ln(0.4142\dots) + \ln(2.4142\dots)$$
$$= -0.8813\dots + 0.8813\dots = 0$$

which is ok.

$$x = -5.4142\dots:$$
$$\ln(-5.4142\dots + 3) + \ln(-5.4142\dots + 5)$$
$$= \ln(-2.4142\dots) + \ln(-0.4142\dots)$$

which is undefined.

The only valid solution is $x = -2.5857\dots$. ◀

Chapter 2 Fundamental Concepts in Number Theory

Numbers are the foundation of mathematics, representing quantities, measurements, and relationships in both abstract and concrete forms. The study of numbers dates back thousands of years, with early civilisations like the Babylonians, Egyptians, and Greeks laying the groundwork for modern number theory. These ancient cultures developed basic arithmetic, including addition, subtraction, multiplication, and division, which are still taught today.

Number theory, often called the "Queen of Mathematics," is a branch of mathematics devoted to the study of integers and the relationships between them. It encompasses a wide range of topics, including prime numbers, divisibility, modular arithmetic, and the properties of number systems. Number theory has a rich history, with significant contributions from mathematicians such as Euclid, who proved the infinitude of prime numbers around 300 BCE, and Pierre de Fermat, known for Fermat's Last Theorem, a problem that puzzled mathematicians for over 350 years until it was finally solved by Andrew Wiles in 1994.

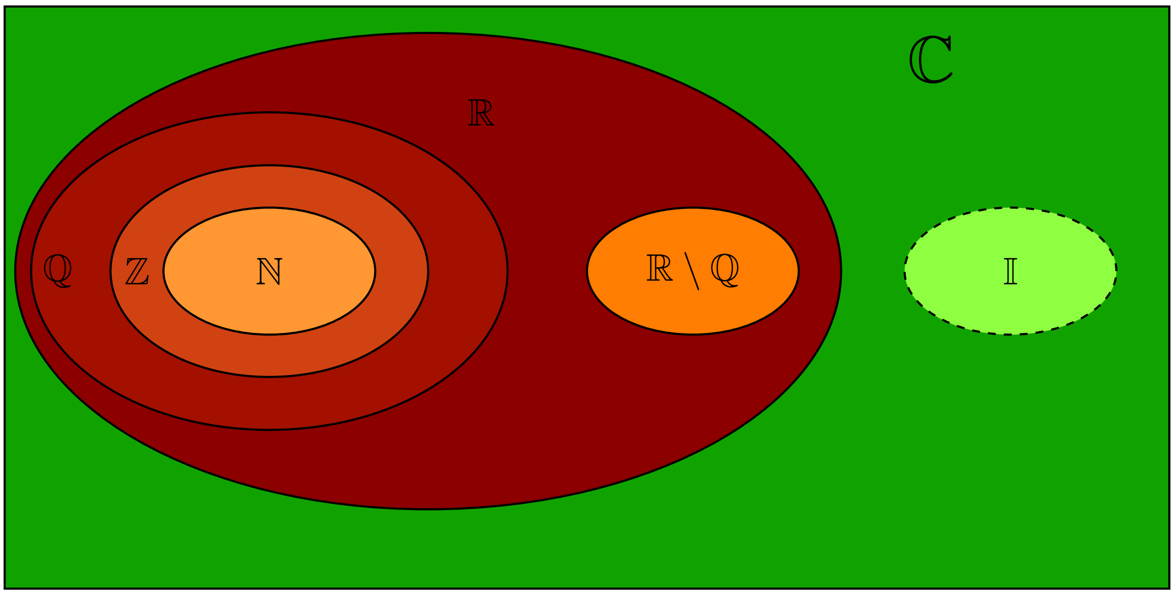


Figure 2.1: Venn diagram of numbers

Throughout history, number theory has evolved from a purely theoretical pursuit to a field with practical applications in modern technology. For example, cryptography, the science of securing communication, relies heavily on number theory, particularly the properties of prime numbers and modular arithmetic. The algorithms that protect our online transactions and digital communications are built upon the principles of number theory.

In this chapter, we will explore various aspects of numbers and number theory, starting with the basics of integers, prime numbers, and number systems, and gradually advancing to more complex topics. By understanding the fundamental properties of numbers, we gain insights into the mathematical structures that underpin the digital world and beyond.

Remark: Many numbers are included in more than one set. Table 2.1 offers an overview of the names, properties of and symbols used for the main number types.

Table 2.1: Types of Numbers and Their Properties

Natural Numbers	$\mathbb{N}$	Numbers used for counting (all positive integers).	$0, 1, 2, \ldots$
Integers	$\mathbb{Z}$	All positive and negative whole numbers.	$\{\ldots, -2, -1, 0, 1, 2, \ldots\}$
Rational Numbers	$\mathbb{Q}$	All real numbers which can be expressed as a fraction, $\frac{p}{q}$, where p and q are integers and $q \neq 0$. All integers are rational numbers as 1 is a non-zero integer.	$\frac{1}{5}, \frac{5}{1}(=5), \frac{2}{3}, \frac{3}{2}, \frac{0}{3}(=0)$
Irrational Numbers	$\mathbb{R} \setminus \mathbb{Q}$	All real numbers which cannot be expressed as a fraction whose numerator and denominator are integers (i.e., all real numbers which aren't rational).	$\pi, \sqrt{2}, \sqrt{3}$
Real Numbers	$\mathbb{R}$	Includes all numbers on the number line.	$\frac{1}{5}, \sqrt{\frac{1}{5}}, 0, -2$
Imaginary Numbers	$\mathbb{I}$	Numbers which are the product of a real number and the imaginary unit i (where $i = \sqrt{-1}$).	$3i = \sqrt{-9},\ -5i = \sqrt{-25},$ $3\sqrt{2}i = \sqrt{-18}$
Complex Numbers	$\mathbb{C}$	All numbers which can be expressed in the form $a + bi$ where a and b are real numbers and $i = \sqrt{-1}$. Each complex number is a combination of a real number (a) and an imaginary number (bi).	$1 + 2i, 1, i, -3i, 0, -5 + i$

Remark: Figure 2.1 may appear misleading at first glance, as it suggests that there are real numbers which are neither rational nor irrational (depicted by the blue region). However, this is not the case. The set of real numbers is exclusively comprised of rational and irrational numbers. This is precisely why the set of irrational numbers is denoted as $\mathbb{R}\setminus\mathbb{Q}$, meaning "all real numbers except the rational numbers". The same is true of the complex numbers; the are all composed of the real numbers and the imaginary numbers.

2.1 Types of Numbers

Let's explore the various types of numbers, including natural numbers, whole numbers, integers, rational numbers, irrational numbers, and real numbers - imaginary numbers and complex numbers are left out of this discussion.

Natural Numbers

We begin with the natural numbers. We distinguish between whole numbers: $0, 1, 2, 3, \ldots$ and counting numbers: $1, 2, 3, \ldots$. These numbers are primarily used for counting. Natural numbers are often regarded as exact values (e.g., there are 4 tires on a car, 8 legs on a spider). However, in some contexts, they may be used as approximations (e.g., there were approximately 1000 people in the crowd).

One of the key properties of natural numbers is that they are *closed* under certain operations, such as addition and multiplication. This means that if you take any two natural numbers and add or multiply them, the result will always be another natural number. For example, if a and b are natural numbers, then both $a + b$ and $a \times b$ are also natural numbers.

There is very little consensus as to whether the symbol $\mathbb{N}$ includes 0. Therefore, the set of whole numbers (that include 0) is often denoted $\mathbb{W}$.

Integers

Integers are the basic building blocks of number theory, consisting of the set of whole numbers and their negatives. Formally, integers include numbers like $\ldots, -3, -2, -1, 0, 1, 2, 3, \ldots$ Basic arithmetic operations, such as addition and multiplication, follow certain properties:

Commutative, Associative, and Distributive Laws

- Commutativity: $a + b = b + a$
- Associativity: $(a + b) + c = a + (b + c)$
- Distributive property: $a \times (b + c) = a \times b + a \times c$

Integers are a fundamental set of numbers in mathematics, denoted by the symbol $\mathbb{Z}$. This set includes all the positive and negative whole numbers, as well as zero. Integers extend the natural numbers by incorporating their additive inverses, thereby allowing for the complete operation of subtraction within the set.

Additive Identity: The number 0 is known as the additive identity. This is because for any integer a, adding zero does not change the value of a. In mathematical terms, this property is expressed as $0 + a = a + 0 = a$. This identity is crucial because it ensures that the set of integers remains stable under addition.

Multiplicative Identity: Similarly, the number 1 serves as the multiplicative identity. For any integer a, multiplying by one leaves the value of a unchanged: $1 \times a = a \times 1 = a$. This property underpins the stability of integers under multiplication, maintaining the integrity of the set.

Additive Inverse: For each integer a, there exists a corresponding additive inverse, denoted as $-a$. The additive inverse is defined such that when a and $-a$ are added together, the result is the additive identity, zero: $a + (-a) = 0$. This property allows for the operation of subtraction within the set of integers, as subtraction can be viewed as the addition of an additive inverse.

Closure Property

Addition: The set of integers is closed under the operation of addition. This means that if you take any two integers and add them together, the sum will always be an integer. For example, if a and b are integers, then $a + b$ is also an integer. This closure property ensures that the set of integers is stable and complete under addition, meaning that no matter how many times you add integers together, the result will remain within the set of integers.

Multiplication: The set of integers is also closed under multiplication. If you multiply any two integers, the product will always be an integer. For instance, if a and b are integers, then $a \times b$ is also an integer. This property guarantees that the operation of multiplication, like addition, does not produce results outside the set of integers, thereby preserving the integrity of the set under multiplication.

It is universally agreed upon that the definition of an integer is clear and precise. Therefore, when in doubt, it is advisable to refer to numbers within this set as "integers." When you specifically need to refer to only the positive integers, it is both accurate and professional to explicitly state "positive integers." This terminology not only ensures clarity but also reflects a sound understanding of mathematical conventions.

Remember, zero is neither positive nor negative, so when discussing subsets of integers, careful consideration of this fact is necessary:

- **Integers:** $\mathbb{Z} = \{\ldots, -4, -3, -2, -1, 0, 1, 2, 3, 4, \ldots\}$
- **Negative Integers:** $\mathbb{Z}^- = \{\ldots, -4, -3, -2, -1\}$
- **Positive Integers:** $\mathbb{Z}^+ = \{1, 2, 3, 4, \ldots\}$
- **Non-Negative Integers:** $\mathbb{Z}_0^+ = \mathbb{Z}_{\geq 0} = \{0, 1, 2, 3, 4, \ldots\}$
- **Non-Positive Integers:** $\mathbb{Z}_0^- = \mathbb{Z}_{\leq 0} = \{\ldots, -4, -3, -2, -1, 0\}$

This notation should be consistent with standard mathematical conventions.

Rational Numbers

Rational numbers, denoted by the symbol $\mathbb{Q}$, are numbers that can be expressed as the quotient of two integers, where the numerator is any integer and the denominator is a non-zero integer. Formally, a rational number can be written as $\frac{a}{b}$, where $a \in \mathbb{Z}$ and $b \in \mathbb{Z} \setminus \{0\}$. This set of numbers is fundamental in mathematics as it provides a means to represent fractions and ratios, allowing for a wide range of arithmetic operations, and as such are a generalisation of common fractions which we saw in chapter 1. They can represent any number that can be written as a finite or repeating decimal.

One of the key properties of rational numbers is that each non-zero rational number has a *multiplicative inverse*. The multiplicative inverse of a rational number $\frac{a}{b}$ is the rational number $\frac{b}{a}$, provided that $a \neq 0$. The importance of the multiplicative inverse lies in the fact that when a number is multiplied by its inverse, the result is the *multiplicative identity*, which is 1. In other words, for any rational number $\frac{a}{b}$, its inverse is denoted by $\left(\frac{a}{b}\right)^{-1} = \frac{b}{a}$, and we have:

$$\frac{a}{b} \times \frac{b}{a} = \frac{ab}{ba} = 1$$

This property ensures that rational numbers are closed under multiplication and division (except division by zero), making them a robust and versatile set of numbers for various mathematical operations.

Rational numbers are dense on the number line, meaning that between any two rational numbers, there exists another rational number. This property makes the set of rational numbers particularly important in the study of real numbers, as it allows for the approximation of irrational numbers to any desired degree of accuracy.

Irrational Numbers

Irrational numbers, denoted by the symbol $\mathbb{R}\setminus\mathbb{Q}$, are real numbers that cannot be expressed as the quotient of two integers. Unlike rational numbers, which have a repeating or terminating decimal expansion, the decimal expansion of an irrational number neither repeats nor terminates. This property makes irrational numbers fundamentally different from rational numbers, as they cannot be precisely represented as fractions.

Some of the most famous examples of irrational numbers include:

- **Pi** (π): Perhaps the most well-known irrational number, π is the ratio of the circumference of a circle to its diameter. Its value is approximately $3.14159\ldots$, but its decimal expansion goes on infinitely without repeating. π plays a crucial role in geometry, trigonometry, and calculus.
- **The Golden Ratio** (ϕ): The golden ratio, $\phi \approx 1.61803\ldots$, is an irrational number that appears frequently in nature, art, and architecture. It is defined as the positive solution to the equation $x^2 - x - 1 = 0$, and it is the limit of the ratio of successive Fibonacci numbers.
- **The Square Root of 2** ($\sqrt{2}$): The square root of 2, approximately $1.41421\ldots$, is the length of the diagonal of a square with side length 1. This number is historically significant because its discovery by the ancient Greeks, particularly the Pythagoreans, revealed the existence of numbers that could not be expressed as the ratio of two integers. The Pythagorean theorem states that in a right triangle, the square of the hypotenuse is equal to the sum of the squares of the other two sides. For a right triangle with both legs of length 1, the hypotenuse is $\sqrt{2}$, demonstrating that $\sqrt{2}$ cannot be a rational number.
- **Euler's Number** (e): The number $e \approx 2.71828\ldots$ is the base of the natural logarithm and is a fundamental constant in mathematics, particularly in calculus and complex analysis. The number e arises naturally in various growth processes, such as compound interest and population growth.

Irrational numbers fill the gaps between rational numbers on the number line, making the real numbers a continuous set. However, like rational numbers, they too have important properties. For instance, while irrational numbers do not have a simple fractional representation, they are nonetheless essential in representing the lengths, areas, and volumes that cannot be captured by rational numbers alone.

Real Numbers

Real numbers, denoted by the symbol $\mathbb{R}$, form the foundation of most mathematical analysis and are essential in describing continuous quantities. The set of real numbers is composed of both rational numbers ($\mathbb{Q}$) and irrational numbers ($\mathbb{R}\setminus\mathbb{Q}$), thus encompassing all numbers that can be placed on the number line. While both rational and irrational numbers are part of the real number system, they differ significantly in their properties:

Similarities and Differences of Rational and Irrational Numbers

- **Representation:** Rational numbers can be represented as fractions, while irrational numbers cannot. This distinction makes irrational numbers more complex to handle, especially in arithmetic operations.
- **Decimal Expansion:** The decimal expansion of rational numbers is either finite or periodic, while that of irrational numbers is infinite and non-repeating.
- **Closure Properties:** Rational numbers are closed under addition, subtraction, multiplication, and division (except by zero). Irrational numbers are not closed under these operations; for example, the sum or product of two irrational numbers can sometimes be rational.
- **Density:** Rational numbers are dense on the real number line, meaning that between any two rational numbers, there exists another rational number. Irrational numbers are also densely distributed but in a complementary manner, filling in the "gaps" left by the rationals, ensuring that the real number line is continuous without any breaks.

The real number line is a continuous, unbroken line that extends infinitely in both directions. Every point on this line corresponds to a unique real number, whether rational or irrational. This continuity is what allows the real numbers to model continuous phenomena in nature, such as time, distance, and temperature.

2.2 Integer Properties and Modular Arithmetic

In 1 we introduced the concept of a factor. A factor is a portion of a quantity, usually an integer or polynomial that, when multiplied by other factors, gives the entire quantity.

In number theory, a factor of a number n is synonymous with a **divisor** of n. Specifically, a divisor of an integer n is an integer d that divides n and results in another integer.

A positive **proper divisor** is a positive divisor of a number n, excluding n itself. Similarly, a **proper factor** of a positive integer n is a factor of n other than 1 or n.

For example, the factors of 12 are:

$$1, 2, 3, 4, 6, 12$$

In this list, 2, 3, 4, and 6 are proper factors of 12, meaning they are less than 12 itself but greater than 1.

A prime number is a natural number greater than 1 that has no positive divisors other than 1 and itself. In other words, a prime number cannot be formed by multiplying two smaller natural numbers. The first few prime numbers are:

$$2, 3, 5, 7, 11, 13, 17, 19, 23, 29, \ldots$$

Notice that 2 is the only even prime number because any other even number can be divided by 2, making it *composite*. A **composite number**, in contrast to a prime, is a natural number greater than 1 that can be divided evenly by at least one positive integer other than 1 and itself. For example, 6 is a composite number because it can be expressed as $6 = 2 \times 3$.

Prime factorisation

Prime factorisation is the process of breaking down a composite number into a product of its prime factors. For any composite number, this factorisation is unique, except for the order of the factors. For instance, consider the number 60:

$$60 = 2 \times 2 \times 3 \times 5 = 2^2 \times 3 \times 5$$

Here, 2, 3, and 5 are prime factors of 60 and $2^2 \times 3 \times 5$ is the **canonical factorisation** of 60 - we usually just call it the prime factorisation.

Theorem 2.1 (The Fundamental Theorem of Arithmetic)

Every integer greater than 1 can be represented uniquely as a product of prime numbers, up to the order of the factors. ♡

This theorem underscores the importance of prime numbers as the "building blocks" of all natural numbers.

Example 2.1

$$84 = 2^2 \times 3 \times 7$$

No matter how you factorise 84, you will always end up with this set of prime numbers (2, 3, and 7), though the order might differ.

The prime factorisation allows for a unique representation of the number in terms of its prime factors.

Greatest Common Divisor (GCD)

The Greatest Common Divisor (GCD), also referred to as the *highest common factor* (HCF) or *greatest common factor* (GFC), of two or more integers is the largest positive integer that divides each of the given integers without leaving a remainder. To determine the GCD, one commonly employs the prime factorisation of the numbers involved. The GCD is then found by multiplying the common prime factors with the smallest exponents from the prime factorisation of the numbers.

For example, consider finding the GCD of 24 and 36:

$$24 = 2^3 \times 3, \quad 36 = 2^2 \times 3^2$$

The common prime factors are 2 and 3. Thus, the GCD is:

$$\text{GCD} = 2^2 \times 3 = 12$$

Example 2.2 Find the GCD of 36 and 120

Solution:

The arrows indicate the numbers that we chose.

$$36 = \overset{\downarrow}{\textcircled{2^2}} \times \textcircled{3^2}$$

$$120 = \textcircled{2^3} \times \underset{\uparrow}{\textcircled{3}} \times 5$$

We see that 2 and 3 are common factors, and choose the ones with the smallest exponents:

$$GCD = 2^2 \times 3 = 12$$

◀

Least Common Multiple (LCM)

The Least Common Multiple (LCM) of two or more integers is the smallest positive integer that is evenly divisible by each of the given integers. The LCM can also be determined using the prime factorisation of the numbers.

For instance, to find the LCM of 24 and 36:

$$24 = 2^3 \times 3, \quad 36 = 2^2 \times 3^2$$

The LCM is obtained by taking the highest power of each prime factor that appears in the factorisation of either number:

$$\text{LCM} = 2^3 \times 3^2 = 72$$

Example 2.3 Find the LCM of 12 and 18

Solution:

$$15 = 3 \times 5$$

$$18 = 2 \times 3^2$$

So the factors with the highest exponents in either factorisation is 2, 3^2, and 5, so

$$LCM = 2 \times 3^2 \times 5 = 90$$

◀

Example 2.4 Find the LCM of 80 and 120

Solution:

The arrows indicate the numbers that we chose.

$$80 = \overset{\downarrow}{2^4} \times \overset{\downarrow}{5}$$

$$120 = \underset{\uparrow}{2^3} \times 3 \times \underset{\uparrow}{5}$$

We see that 2 and 5 are common factors and choose the ones with the smallest exponents (5 has the same exponents so we just take one of them):

$$LCM = 2^4 \times 3 \times 5 = 240$$

◀

Connection Between GCD and LCM

An important relationship between the GCD and LCM of two numbers a and b is given by the formula:

$$\text{GCD}(a,b) \times \text{LCM}(a,b) = a \times b$$

For example, using the numbers 24 and 36:

$$12 \times 72 = 24 \times 36 = 864$$

This relationship is a powerful tool in solving problems related to divisibility, number theory, and algebra.

The study of primes and factors is foundational in mathematics, providing essential tools for understanding the structure of numbers. The concepts of prime factorisation, the Fundamental Theorem of Arithmetic, and the calculations of GCD and LCM are not only critical in theoretical mathematics but also in practical applications, such as cryptography, coding theory, and the analysis of algorithms.

Divisors and Remainders

Division involving integers can be tricky since the result might not always be an integer. Often, division produces a remainder. For example,

$$9 = 2 \times 4 + 1.$$

In this case, dividing 9 by 4 leaves a *remainder* of 1.

In general, for any integers a and b, we can express this as:

$$b = k \times a + r,$$

where r is the remainder. If r is zero, then we say that a divides b, denoted as $a \mid b$. The vertical bar is used to signify divisibility. For instance, $2 \mid 128$ and $7 \mid 49$, but 3 does not divide 4, which is written as $3 \nmid 4$.

Above, we made no distinction between factors and divisors. Some argue that they differ slightly:

- If $a \mid b$ and $a > 0$, a is a *divisor* of b.
- If $a \notin \{1, b\}$ and $a \mid b$, a is a factor of b.

In this definition, primes have no factors, only two divisors 1 and itself. We will make no further distinction between primes and factors and use the terms interchangeably.

Modular Arithmetic

The `mod` operator, commonly encountered in computer programming, provides the remainder after division. For example:

1. $25 \bmod 4 = 1$ because $25 \div 4 = 6$ with a remainder of 1.
2. $19 \bmod 5 = 4$ since $19 = 3 \times 5 + 4$.
3. $24 \bmod 5 = 4$.
4. $99 \bmod 11 = 0$.

While there are some complexities when dealing with negative numbers, we will focus on positive integers for simplicity. It's also worth noting that the results of the modulus operation are often expressed differently. For instance, $24 = 4 \bmod 5$ or $21 = 0 \bmod 7$, which means $24 \bmod 5 = 4$ and $21 \bmod 7 = 0$.

Modular arithmetic is sometimes referred to as clock arithmetic. Imagine using a 24-hour clock: 09:00 represents 9 in the morning, while 21:00 represents 9 in the evening. If a journey starts at 07:00 and lasts 25 hours, the arrival time would be 08:00 the next day. Mathematically, this can be expressed as $7 + 25 = 32$ and $32 \bmod 24 = 8$. Essentially, we start at 7 and move 25 hours forward on the clock face, landing at 8.

Chapter 3 Numeral Systems

We are so accustomed to working within the decimal system that we often forget it is a relatively recent invention and was once considered revolutionary. It is time to carefully examine how we represent numbers. Typically, we use the decimal system, where a number like 3459 is shorthand for $3 \times 1000 + 4 \times 100 + 5 \times 10 + 9$. The position of each digit is crucial, as it allows us to distinguish between values like 30 and 3. The decimal system is a **positional numeral system**, meaning it has designated positions for units, tens, hundreds, and so forth. Each digit's position implies the multiplier (a power of ten) that should be used with that digit, and each position has a value ten times that of the position to its right.

Notice that we can save space by writing 1000 as 10^3, where the exponent 3 indicates the number of zeros. Thus, $100000 = 10^5$. If the exponent is negative, it represents a fraction, e.g., $10^{-3} = \frac{1}{1000}$. Perhaps the most ingenious aspect of the positional system was the addition of the decimal point, which allows us to include decimal fractions. For example, the number 123.456 is equivalent to:

$$1 \times 100 + 2 \times 10 + 3 \times 1 + 4 \times \frac{1}{10} + 5 \times \frac{1}{100} + 6 \times \frac{1}{1000}.$$

This can be visualised as:

$$\begin{array}{rcccccccccc}
\text{Multiplier:} & \ldots & 10^2 & 10^1 & 10^0 & . & 10^{-1} & 10^{-2} & 10^{-3} & \ldots \\
\text{Digits:} & \ldots & 1 & 2 & 3 & . & 4 & 5 & 6 & \ldots \\
 & & & & & \uparrow & & & & \\
 & & & & & \text{Decimal Point:} & & & &
\end{array}$$

However, there is no inherent reason why we must use powers of 10, or base 10. The Babylonians, for instance, used base 60, and base 12 was very common in medieval Europe. Today, the most widely used numeral systems are summarised table 3.1

Numeral system	Symbols	Base	Additional information
Decimal	0-9	10	-
Binary	0, 1	2	-
Hexadecimal	0-9, A-F	16	$\text{A} \equiv 10, \text{B} \equiv 11, \text{C} \equiv 12, \text{D} \equiv 13, \text{E} \equiv 14, \text{F} \equiv 15$
Octal	0-7	8	-

Table 3.1: Summary of Common Numeral Systems

We begin by focusing on binary which will also receive the most detailed attention in this chapter.

3.1 Binary Numbers

In the binary scale, we express numbers in powers of 2 rather than the 10s of the decimal scale. For some numbers, this is easy. Recall $2^0 = 1$,

As in decimal, we write this with the position of the digit representing the power, the first place after the decimal being the 2^0 position, the next the 2^1, and so on. To convert a decimal number to binary, we can use the `mod` operator.

Decimal number		In powers of 2	Power of 2				Binary number
			3	2	1	0	
8	$=$	2^3	1	0	0	0	1000
7	$=$	$2^2+2^1+2^0$	0	1	1	1	111
6	$=$	2^2+2^1	0	1	1	0	110
5	$=$	2^2+2^0	0	1	0	1	101
4	$=$	2^2	0	1	0	0	100
3	$=$	2^1+2^0	0	0	1	1	11
2	$=$	2^1	0	0	1	0	10
1	$=$	2^0	0	0	0	1	1

Table 3.2: Decimal Numbers in Binary Representation

As an example, consider 88 in decimal or 88_{10}. We would like to write it as a binary number. We take the number and successively divide `mod` 2. See below:

Step Number n	x_n	$x_n/2$	$x_n \bmod 2$
0	88	44	0
1	44	22	0
2	22	11	0
3	11	5	1
4	5	2	1
5	2	1	0
6	1	0	1

Table 3.3: Conversion of Decimal 88 to Binary

Writing the last column in reverse, that is from the bottom up, we have 1011000, which is the binary form of 88, i.e., $88_{10} = 1011000_2$.

Binary decimals are less common but quite possible. Thus, 101.1011 is just $2^2 + 2^0 + 2^{-1} + 2^{-3} + 2^{-4}$, which is, after some calculation, 5.6875. We have seen how to turn the integer part of a decimal number into a binary number, and we can do the same with a decimal fraction. Consider 0.6875. As before, we draw up a table:

Step Number n	x_n	$x_n \times 2$	$\lfloor x_n \times 2 \rfloor$
0	0.6875	1.375	1
1	0.375	0.75	0
2	0.75	1.5	1
3	0.5	1	1

Table 3.4: Conversion of Decimal Fraction 0.6875 to Binary

Giving, reading down, $0.6875_{10} = 1011_2$.

Binary Expansion

The process outlined in the previous section is called **binary expansion** and refers to the representation of a number in the binary (base-2) numeral system. Every decimal number can be expressed as a sum of powers of 2, where each power corresponds to a binary digit (bit) in the number's binary form.

Let's reconsider the decimal number 88. To find its binary expansion, we identify the largest power of 2 less than or equal to 88 and continue subtracting powers of 2 until we reach 0.

First, we note that $2^6 = 64$ is the largest power of 2 less than 88:

$$88 = 64 + 24$$

Next, we find that $2^4 = 16$ is the largest power of 2 less than 24:

$$24 = 16 + 8$$

Finally, $2^3 = 8$ exactly matches the remainder:

$$8 = 8 + 0$$

Thus, we have:

$$88 = 2^6 + 2^4 + 2^3$$

In binary, each of these powers of 2 is represented by a '1' in the corresponding place value, with '0' in place values where no power of 2 contributes:

$$88_{10} = 1011000_2$$

To summarise:

- $2^6 = 64$ corresponds to the leftmost '1' in the binary expansion.
- $2^4 = 16$ corresponds to the next '1'.
- $2^3 = 8$ corresponds to the next '1'.
- The remaining digits are '0' because 2^5, 2^2, 2^1, and 2^0 do not contribute to the value 88.

Thus, the binary expansion of 88 is 1011000_2. This method of representing numbers is fundamental in computer science and digital electronics, where binary representation is the standard for data storage and processing.

Binary Operations

Binary operations are basic arithmetic operations performed on binary numbers. These operations are essential in computing and digital systems, as they form the foundation for how computers process and manipulate data.

Binary addition, subtraction, and multiplication are similar to their decimal counterparts but follow simpler rules due to the binary system's limited digits. For example, binary addition follows these rules:

$$0 + 0 = 0, \quad 0 + 1 = 1, \quad 1 + 0 = 1, \quad 1 + 1 = 10$$

In this case, $1+1$ results in 10_2, which means 0 with a carry of 1 to the next higher bit. Binary subtraction and multiplication follow similar straightforward rules that are easy to implement in digital systems.

The XOR (exclusive OR) operation is another important binary operation. XOR produces a 1 if the two bits being compared are different and a 0 if they are the same:

$$0 \oplus 0 = 0, \quad 0 \oplus 1 = 1, \quad 1 \oplus 0 = 1, \quad 1 \oplus 1 = 0$$

In binary addition, the XOR operation is used to add two bits without considering any carry from a previous bit. This is because XOR effectively performs addition modulo 2, which aligns perfectly with how binary addition works. For example:

Bit 1	Bit 2	XOR (Sum)	AND (Carry)
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

In the case of $1+1$, XOR gives a sum of 0 and an AND operation (which detects the carry) gives a carry of 1, resulting in the binary number 10.

$$\begin{aligned} 0+0&=0 \\ 0+1&=1 \\ 1+1&=10 \quad \text{so we carry 1 and leave a zero} \\ 1+1+1=1+(1+1)=1+10&=11. \end{aligned}$$

We can write this in very much the same way as for a decimal addition:

$$\begin{array}{c|c|c|c|c|c|c|c}
 & 1 & 1 & 0 & 1 & 0 & 1 & \\
+ & 1 & 0 & 1 & 1 & 1 & 0 & \\
\hline
1 & 1 & 0 & 0 & 0 & 1 & 1 & \text{Sum} \\
\uparrow & & & & \uparrow & & & \\
 & & & & & & &
\end{array}$$

The right-hand arrow shows where we carry a 1. The left-hand arrow shows where we have $1+1+1$ so we carry a 1 and have a 1 left over.

As we will see below, we will often need to handle multiple carries. There are two ways to handle this which resemble the methods we know from the decimal system. We will explain using an example.

Method 1: Column-wise Binary Addition with Multiple Carries

Consider

$$\begin{array}{cccccc}
 & 1 & 1 & 1 & 1 & 1 \\
+ & 1 & 1 & 1 & 0 & 1 \\
+ & 1 & 1 & 1 & 0 & 1 \\
+ & 1 & 1 & 1 & 1 & 1 \\
\hline
\end{array}$$

Step 1: Add the Rightmost Column
Start by adding the rightmost bits:

$$1 + 1 + 1 + 1 = 100_2 \quad (\text{ which is binary for } 4)$$

Reading the result from right to left (i.e. from *least significant bit* (LSB) to the *most significant bit*(MSB))

- write down the 0
- carry the 0 to the next column
- carry the 1 to the third column

You end up with

$$\begin{array}{cccccc}
 & & & {\scriptstyle 1} & {\scriptstyle 0} & \\
 & 1 & 1 & 1 & 1 & 1 \\
+ & 1 & 1 & 1 & 0 & 1 \\
+ & 1 & 1 & 1 & 0 & 1 \\
+ & 1 & 1 & 1 & 1 & 1 \\
\hline
 & & & & & 0
\end{array}$$

Step 2: Add the Second Column from the Right
Next, add the second column:

$$0 + 1 + 0 + 0 + 1 = 10_2 \quad (\text{ which is binary for } 2)$$

Reading the result from LSB to MSB:

- write down the 0
- carry the 1 to the next column

You end up with

$$\begin{array}{cccccc}
 & & & {\scriptstyle 1} & & \\
 & & & {\scriptstyle 1} & & \\
 & 1 & 1 & 1 & 1 & 1 \\
+ & 1 & 1 & 1 & 0 & 1 \\
+ & 1 & 1 & 1 & 0 & 1 \\
+ & 1 & 1 & 1 & 1 & 1 \\
\hline
 & & & & 0 & 0
\end{array}$$

Step 3: Add the Third Column from the Right
Now, add the third column:

$$1+1+1+1+1+1 = 110_2 \quad (\text{ which is binary for } 6)$$

Reading the result from LSB to MSB

- write down the 0
- carry the 1 to the fourth column
- carry the 1 to the fifth column

You end up with

Our sum so far:

$$\begin{array}{rccccc}
 & {\scriptstyle 1} & {\scriptstyle 1} & & & \\
 & 1 & 1 & 1 & 1 & 1 \\
+ & 1 & 1 & 1 & 0 & 1 \\
+ & 1 & 1 & 1 & 0 & 1 \\
+ & 1 & 1 & 1 & 1 & 1 \\ \hline
 & & & 0 & 0 & 0
\end{array}$$

Step 4: Add the Fourth Column from the Right
Move to the fourth column:

$$1+1+1+1+1 = 101_2 \quad (\text{ which is binary for } 5)$$

Reading the result from LSB to MSB

- write down the 1
- carry the 0 to the fifth column
- carry the 1 to the sixth column

You end up with

$$\begin{array}{rcccccc}
 & & {\scriptstyle 0} & & & & \\
 & {\scriptstyle 1} & {\scriptstyle 1} & & & & \\
 & & 1 & 1 & 1 & 1 & 1 \\
+ & & 1 & 1 & 1 & 0 & 1 \\
+ & & 1 & 1 & 1 & 0 & 1 \\
+ & & 1 & 1 & 1 & 1 & 1 \\ \hline
 & & & 1 & 0 & 0 & 0
\end{array}$$

Step 5: Add the Leftmost Column
Add the leftmost column:

$$1+1+1+1+1 = 101_2 \quad (\text{ which is binary for } 5)$$

Reading the result from LSB to MSB

- write down the 1
- carry the 0 to the sixth column
- carry the 1 to the seventh column

This results in

$$\begin{array}{cccccccc}
 & & \scriptstyle 0 & & & & & \\
 & \scriptstyle 1 & \scriptstyle 1 & & & & & \\
 & & & 1 & 1 & 1 & 1 & 1 \\
+ & & & 1 & 1 & 1 & 0 & 1 \\
+ & & & 1 & 1 & 1 & 0 & 1 \\
+ & & & 1 & 1 & 1 & 1 & 1 \\
\hline
 & & & 1 & 1 & 0 & 0 & 0
\end{array}$$

Step 6: Add the Remaining Carries
Finally, add the remaining carries:

$$\begin{array}{cccccccc}
 & & & 1 & 1 & 1 & 1 & 1 \\
+ & & & 1 & 1 & 1 & 0 & 1 \\
+ & & & 1 & 1 & 1 & 0 & 1 \\
+ & & & 1 & 1 & 1 & 1 & 1 \\
\hline
 & 1 & 1 & 1 & 1 & 0 & 0 & 0
\end{array}$$

The following example demonstrates the entire process by using different colors to distinguish each column and the corresponding carries they produce. Note that the last two digits in the sum are colored black, as they do not result from any specific column but are instead generated solely from the carries.

$$\begin{array}{cccccccc}
 & \scriptstyle 1 & \scriptstyle 0 & & & & & \\
 & & \scriptstyle 1 & \scriptstyle 0 & & \scriptstyle 1 & & \\
 & & & \scriptstyle 1 & \scriptstyle 1 & \scriptstyle 1 & \scriptstyle 0 & \\
 & & & 1 & 1 & 1 & 1 & 1 \\
+ & & & 1 & 1 & 1 & 0 & 1 \\
+ & & & 1 & 1 & 1 & 0 & 1 \\
+ & & & 1 & 1 & 1 & 1 & 1 \\
\hline
 & 1 & 1 & 1 & 1 & 0 & 0 & 1
\end{array}$$

Method 2: Direct Summation and Simplification

We will illustrate the second method using the same example. In the previous case, we carried the actual binary number to the next columns. In this method, we write down 0 if the sum is even and 1 if the sum is odd. Every time a sum a multiple of 2, we carry a 1 to the next columns, and then continue this process for each column, including the carries in the calculation of that column.

Step 1: Add the Rightmost Column
Add bits in column 1 (from counting from MSB):

$$1+1+1+1=100_2 \quad (\text{ which is binary for } 4)$$

Reading the result from LSB to MSB

- write down the 0
- carry a 1 for the first multiple of 2
- carry a 1 for the second multiple of 2

This results in

$$\begin{array}{r@{\qquad}ccccc}
 & & & & {\scriptstyle 1} & \\
 & & & & {\scriptstyle 1} & \\
 & 1 & 1 & 1 & 1 & 1 \\
+ & 1 & 1 & 1 & 0 & 1 \\
+ & 1 & 1 & 1 & 0 & 1 \\
+ & 1 & 1 & 1 & 1 & 1 \\ \hline
 & & & & & 0
\end{array}$$

Step 2: Add the Second Column from the Right
Add bits in column 2 (from counting from MSB):

$$1+1+1+1=100_2 \quad (\text{ which is binary for } 4)$$

Reading the result from LSB to MSB

- write down the 0
- carry a 1 for the first multiple of 2
- carry a 1 for the second multiple of 2

This results in

$$\begin{array}{r@{\qquad}ccccc}
 & & & {\scriptstyle 1} & & \\
 & & & {\scriptstyle 1} & & \\
 & 1 & 1 & 1 & 1 & 1 \\
+ & 1 & 1 & 1 & 0 & 1 \\
+ & 1 & 1 & 1 & 0 & 1 \\
+ & 1 & 1 & 1 & 1 & 1 \\ \hline
 & & & & 0 & 0
\end{array}$$

Step 3: Add the Third Column from the Right Add bits in column 3 (from counting from MSB):

$$1+1+1+1+1+1 = 110_2 \quad (\text{ which is binary for } 6)$$

Reading the result from LSB to MSB

- write down the 0
- carry a 1 for the first multiple of 2
- carry a 1 for the second multiple of 2
- carry a 1 for the third multiple of 2

This results in

$$\begin{array}{cccccc}
 & & 1 & & & \\
 & & 1 & & & \\
 & & 1 & & & \\
 & 1 & 1 & 1 & 1 & 1 \\
+ & 1 & 1 & 1 & 0 & 1 \\
+ & 1 & 1 & 1 & 0 & 1 \\
+ & 1 & 1 & 1 & 1 & 1 \\
\hline
 & & & 0 & 0 & 0
\end{array}$$

Step 4: Add the Fourth Column from the Right
Add bits in column 4 (from counting from MSB):

$$1+1+1+1+1+1+1 = 111_2 \quad (\text{ which is binary for } 7)$$

Reading the result from LSB to MSB

- write down the 1
- carry a 1 for the first multiple of 2
- carry a 1 for the second multiple of 2
- carry a 1 for the third multiple of 2

This results in

$$\begin{array}{cccccc}
 & 1 & & & & \\
 & 1 & & & & \\
 & 1 & & & & \\
 & 1 & 1 & 1 & 1 & 1 \\
+ & 1 & 1 & 1 & 0 & 1 \\
+ & 1 & 1 & 1 & 0 & 1 \\
+ & 1 & 1 & 1 & 1 & 1 \\
\hline
 & & 1 & 0 & 0 & 0
\end{array}$$

Step 5: Add the Leftmost Column
Add bits in leftmost column):

$$1+1+1+1+1+1+1 = 111_2 \quad \text{(which is binary for 7)}$$

Reading the result from LSB to MSB

- write down the result in binary, i.e. 1 1 1

This results in

$$\begin{array}{ccccccccc}
 & & & 1 & 1 & 1 & 1 & 1 \\
+ & & & 1 & 1 & 1 & 0 & 1 \\
+ & & & 1 & 1 & 1 & 0 & 1 \\
+ & & & 1 & 1 & 1 & 1 & 1 \\
\hline
 & 1 & 1 & 1 & 1 & 0 & 0 & 0
\end{array}$$

which corresponds to the result we obtained above. While not demonstrated explicitly here, subtraction works in a similar fashion.

By using one of these methods for handling multiple carries, allow us to also multiply two binary numbers.

Multiplication in Binary

Multiplication in binary is technically easier than multiplication in decimal. In binary operations, we work exclusively with two digits: 0 and 1. This means that both the the multiplier[1] and multiplicand consist of 0's and 1' (and so does the multiplicand). The process of finding the binary product is analogous to traditional multiplication in the decimal system. The four five steps involved in multiplying binary digits are:

$0 \times 0 = 0$

$0 \times 1 = 0$

$1 \times 0 = 0$

$1 \times 1 = 1$

$1 \times 10_2 = 10_2$ (multiplying by base 10_2 adds a 0 to the end)

The last step means that $101_2 \times 10_2 = 1010_2$ which is analogous to the decimal case: $143_{10} \times 10_{10} = 1430_{10}$.

We will illustrate the process by supplying a couple of examples.

[1]The "multiplicand" is the number that has to be multiplied, and the "multiplier" is the number by which it is multiplied.

Example 3.1

$$\begin{array}{cl rrrr}
 & (1) & & 1 & 0 & 0 \\
\times & (2) & & & 1 & 1 \\
\hdashline
 & (3) & & 1 & 0 & 0 \\
+ & (4) & 1 & 0 & 0 & 0 \\
\hline
 & (5) & 1 & 1 & 0 & 0
\end{array}$$

Here are the steps:

- Multiply the multiplicand (line 1) by the LSB of the multiplier (line 2), which in this case is 1.
- Record this result in line 3.
- Append a 0 to line 4 to account for the shift to the next power of 2 in the multiplier.
- Multiply the multiplicand (line 1) by the next bit of the multiplier (line 2), which is also 1 in this case.
- Add this result to line 4, after the 0 you appended earlier.
- Finally, sum the values in lines 3 and 4, as outlined in the previous section, to obtain the final result in line 5.

We offer two additional examples:

Example 3.2

$$\begin{array}{c rrrrrr}
 & & & & 1 & 0 & 1 \\
\times & & & 1 & 0 & 1 & 1 \\
\hdashline
 & & & & 1 & 0 & 1 \\
+ & & & 1 & 0 & 1 & 0 \\
+ & & 0 & 0 & 0 & 0 & 0 \\
+ & 1 & 0 & 1 & 0 & 0 & 0 \\
\hline
 & 1 & 1 & 0 & 1 & 1 & 1
\end{array}$$

Example 3.3

$$\begin{array}{c rrrrrrrrr}
 & & & 1 & 0 & 0 & 1 & 1 & 1 & 0 \\
\times & & & & & & & 1 & 0 & 1 \\
\hdashline
 & & & 1 & 0 & 0 & 1 & 1 & 1 & 0 \\
+ & 1 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 \\
\hline
 & 1 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 0
\end{array}$$

In example 3.3 notice that we omitted the row of zeroes that the second value of the multiplier would have produced, and notice even further that we added two 0's before restating the multiplicand in the sum.

Binary multiplication, like binary addition, is a core operation in computer arithmetic. By breaking the process down into manageable steps—multiplying individual bits and then summing the results—it becomes clear how similar it is to the multiplication methods we use in the decimal system. The main difference is the simplicity and efficiency of working within the binary system, where only the digits 0 and 1 are involved.

We notice how binary multiplication builds on binary addition. Each step, involving shifts and sums, essentially consists of repeated additions adjusted by powers of two. A strong grasp of binary addition naturally leads to a better understanding of binary multiplication and its applications.

3.2 Octal and Hexadecimal

Octal is a base-8 numbering system that uses the digits 0 through 7. It is closely related to binary, which is a base-2 system. The connection between the two lies in how easily binary numbers can be converted to octal and vice versa. Each octal digit corresponds to exactly three binary digits (bits), making conversions straightforward. For example, the binary number '110' converts directly to the octal digit '6'. Because of this close relationship, octal is often used as a shorthand for binary in computing, particularly in contexts where grouping binary digits in sets of three simplifies reading and interpreting binary data.

$$12_8 = 1 \cdot 8^1 + 2 \cdot 8^0 = 10_{10}$$
$$3021_8 = 3 \cdot 8^3 + 0 \cdot 8^2 + 2 \cdot 8^1 + 1 \cdot 8^0 = 1553_{10}$$

Since 8 is 2^3, we can express it in binary:

$3 \rightarrow 011$

$0 \rightarrow 000$

$2 \rightarrow 010$

$1 \rightarrow 001$

Thus, $3021_8 = 011000010001_2 = 11000010001_2$

We obtain the final result by removing leading zeros.

Hexadecimal is a base-16 numbering system that uses sixteen distinct symbols: the digits 0-9 and the letters A-F, where A represents 10, B represents 11, and so on up to F, which represents 15. Hexadecimal is closely related to binary because each hexadecimal digit corresponds exactly to four binary digits (bits). This direct relationship makes it easy to convert between the two systems. For example, the hexadecimal digit 'A' translates to the binary sequence '1010'. Due to this efficiency in grouping, hexadecimal is often used in computing as a more compact and readable way to represent binary data, particularly in areas like memory addresses and colour codes in web design.

$$123_{16} = 1 \cdot 16^2 + 2 \cdot 16^1 + 3 \cdot 16^0 = 256 + 32 + 3 = 291_{10}$$
$$A2E_{16} = 10 \cdot 16^2 + 2 \cdot 16^1 + 14 \cdot 16^0 = 2560 + 32 + 14 = 2606_{10}$$

Since 16 is 2^4, we can, for instance, express $A2E_{16}$ in binary:

$A \rightarrow 1010$

$2 \rightarrow 0010$

$E \rightarrow 1110$

Thus, $A2E_{16} = 101000101110_2$

Similarly we get $5EB52_{16}$ as

$$\begin{aligned}
5 &\to 0101\\
E &\to 1110\\
B &\to 1011\\
5 &\to 0101\\
2 &\to 0010
\end{aligned}$$

Thus, $5EB52_{16} = 01011110101101010010_2$

Again, notice that we removed the leading 0's from 5_{16} when writing the result.

3.3 Converting Between Systems

Understanding how to convert numbers between binary, decimal, octal, and hexadecimal systems is essential in computer science and digital electronics. Each system is a different base, and each has its own applications. Here's a step-by-step guide to help you convert numbers from one system to another.

Decimal to Binary Conversion

To convert a decimal number to binary:

1. Divide the decimal number by 2.
2. Record the remainder (it will be 0 or 1).
3. Divide the quotient by 2 and record the remainder.
4. Repeat until the quotient is 0.
5. The binary number is the sequence of remainders read from bottom to top.

Example 3.4 Convert 23_{10} to binary.

Solution:

$$\begin{aligned}
23 \div 2 &= 11 \quad \text{remainder } 1\\
11 \div 2 &= 5 \quad \text{remainder } 1\\
5 \div 2 &= 2 \quad \text{remainder } 1\\
2 \div 2 &= 1 \quad \text{remainder } 0\\
1 \div 2 &= 0 \quad \text{remainder } 1
\end{aligned}$$

Thus, $23_{10} = 10111_2$. ◀

Decimal to Octal Conversion

To convert a decimal number to octal:

1. Divide the decimal number by 8.
2. Record the remainder.
3. Divide the quotient by 8 and record the remainder.
4. Repeat until the quotient is 0.

5. The octal number is the sequence of remainders read from bottom to top.

Example 3.5 Convert 78_{10} to octal.

Solution:

$$\begin{aligned} 78 \div 8 &= 9 \quad \text{remainder } 6 \\ 9 \div 8 &= 1 \quad \text{remainder } 1 \\ 1 \div 8 &= 0 \quad \text{remainder } 1 \end{aligned}$$

Thus, $78_{10} = 116_8$. ◀

Decimal to Hexadecimal Conversion

To convert a decimal number to hexadecimal:

1. Divide the decimal number by 16.
2. Record the remainder (use A, B, C, D, E, F for remainders 10, 11, 12, 13, 14, 15 respectively).
3. Divide the quotient by 16 and record the remainder.
4. Repeat until the quotient is 0.
5. The hexadecimal number is the sequence of remainders read from bottom to top.

Example 3.6 Convert 255_{10} to hexadecimal.

Solution:

$$\begin{aligned} 255 \div 16 &= 15 \quad \text{remainder } 15 \quad (\text{F}) \\ 15 \div 16 &= 0 \quad \text{remainder } 15 \quad (\text{F}) \end{aligned}$$

Thus, $255_{10} = FF_{16}$. ◀

Binary to Decimal Conversion

To convert a binary number to decimal:

1. Multiply each bit by 2 raised to the power of its position, starting from 0 on the right.
2. Sum all the products.

Notice that this amounts to the method outlined above about binary expansion.

Example 3.7 Convert 1101_2 to decimal.

Solution:

$$1 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 = 8 + 4 + 0 + 1 = 13_{10}$$

◀

Binary to Octal Conversion

To convert a binary number to octal:

1. Group the binary digits into sets of three, starting from the right. Add leading zeros if necessary.
2. Convert each group of three binary digits to its octal equivalent.

Example 3.8 Convert 110110_2 to octal.

Solution:

$110 \rightarrow 6$

$110 \rightarrow 6$

Thus, $110110_2 = 66_8$. ◀

Binary to Hexadecimal Conversion

To convert a binary number to hexadecimal:

1. Group the binary digits into sets of four, starting from the right. Add leading zeros if necessary.
2. Convert each group of four binary digits to its hexadecimal equivalent.

Example 3.9 Convert 10110101_2 to hexadecimal.

Solution:

$1011 \rightarrow B$

$0101 \rightarrow 5$

Thus, $10110101_2 = B5_{16}$. ◀

Octal to Binary Conversion

To convert an octal number to binary:

1. Convert each octal digit to its 3-bit binary equivalent.

Example 3.10 Convert 57_8 to binary.

Solution:

$5 \rightarrow 101$

$7 \rightarrow 111$

Thus, $57_8 = 101111_2$. ◀

Octal to Decimal Conversion

To convert an octal number to decimal:

1. Multiply each digit by 8 raised to the power of its position, starting from 0 on the right.
2. Sum all the products.

Example 3.11 Convert 157_8 to decimal.

Solution:

$$1 \cdot 8^2 + 5 \cdot 8^1 + 7 \cdot 8^0 = 64 + 40 + 7 = 111_{10}$$

◀

Octal to Hexadecimal Conversion

To convert an octal number to hexadecimal:

1. First, convert the octal number to binary.
2. Then, convert the binary number to hexadecimal by grouping the binary digits in sets of four.

Example 3.12 Convert 157_8 to hexadecimal.

Solution:

$$1 \to 001$$
$$5 \to 101$$
$$7 \to 111$$

Thus, $157_8 = 001101111_2 = 6F_{16}$. ◀

Hexadecimal to Binary Conversion

To convert a hexadecimal number to binary:

1. Convert each hexadecimal digit to its 4-bit binary equivalent.

Example 3.13 Convert $2B_{16}$ to binary.

Solution:

$$2 \to 0010$$
$$B \to 1011$$

Thus, $2B_{16} = 00101011_2$. ◀

Hexadecimal to Decimal Conversion

To convert a hexadecimal number to decimal:

1. Multiply each digit by 16 raised to the power of its position, starting from 0 on the right.
2. Sum all the products.

Example 3.14 Convert $2B_{16}$ to decimal.

Solution:

$$2 \cdot 16^1 + 11 \cdot 16^0 = 32 + 11 = 43_{10}$$

◀

Hexadecimal to Octal Conversion

To convert a hexadecimal number to octal:

1. First, convert the hexadecimal number to binary.
2. Then, convert the binary number to octal by grouping the binary digits in sets of three.

Example 3.15 Convert $2B_16$ to octal.

Solution:

$$\begin{aligned} 2 &\to 0010 \\ B &\to 1011 \end{aligned}$$

Thus, $2B_{16} = 00101011_2 = 53_8$. ◀

Final Thoughts on Conversion

The concept of expansion plays a central role in these conversions. Whether you are expanding a decimal number into its binary, octal, or hexadecimal form, or converting a binary number into its octal or hexadecimal equivalent, you are more or less expressing the number in terms of powers of the base. The expansion method is essentially the same for each system as it boils down to dividing by the highest power of the base recursively:

$$\begin{aligned} 7562_{10} &= 1 \cdot 16^3 + 3466 = 1 \cdot 16^3 + 13 \cdot 16^2 + 138 \\ &= 1 \cdot 16^3 + 13 \cdot 16^2 + 8 \cdot 16^1 + 10 \cdot 16^0 = 1\,D\,8\,A \end{aligned}$$

By understanding these expansions and the relationships between these number systems, you can efficiently switch between them, allowing you to represent and manipulate data in the most suitable format for any given situation.

Chapter 4 Boolean Algebra

In the previous chapter, we explored various numeral systems — binary, decimal, octal, and hexadecimal — and learned how to perform conversions and binary arithmetic. Binary, with its base-2 structure, is particularly significant in the world of computing because all digital systems operate using binary data. Every number and operation inside a computer is ultimately broken down into a series of 0s and 1s.

As we move from working with binary numbers to understanding **Boolean algebra**, we take the next step in comprehending how computers perform logical operations using binary values. While numeral systems allowed us to understand how numbers are represented and manipulated, Boolean algebra enables us to understand how decisions and logical processes are carried out within digital systems. In essence, Boolean algebra provides the rules that guide logical decision-making in binary, the same way arithmetic operates in numeral systems.

4.1 Introduction to Boolean Algebra

Boolean algebra is a mathematical framework that deals with binary values — typically represented as 0 and 1 - and shows us what kind of operations that can be carried out on bits. This system, first introduced by **George Boole** in the mid-19th century, forms the foundation of modern digital logic and is essential in the fields of computer science and electrical engineering.

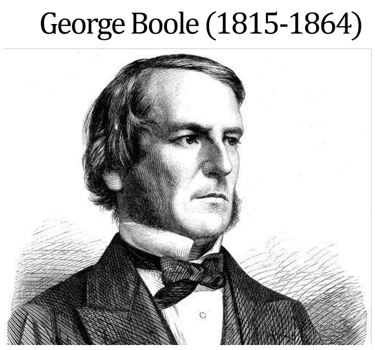


Figure 4.1: Developed the rules of logic.

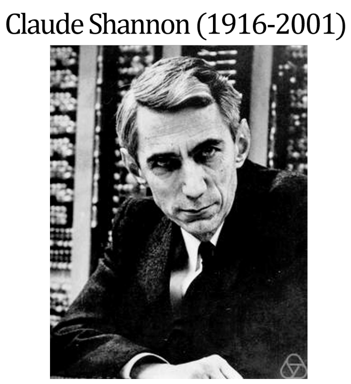


Figure 4.2: Showed how to use Boolean algebra for designing computer circuits.

Although Boole laid the groundwork, it was not until the 1930s that **Claude Shannon**, widely regarded as the father of information theory, demonstrated how Boolean algebra could be applied to the design of electronic circuits. In his groundbreaking master's thesis, Shannon showed that Boolean algebra could be used to model and simplify the complex arrangement of switches and relays found in electrical circuits. This revelation bridged the gap between abstract mathematical logic and practical engineering, laying the foundation for the digital circuits used in modern computers. Shannon's master thesis won the 1939 Alfred Noble Prize.

Relevance to Computer Science

Boolean algebra plays a central role in computer science, especially in the design and functioning of digital systems, such as computers, mobile devices, and embedded systems. At its core, Boolean algebra is used to model and design the behaviour of **digital circuits**, which are built from electronic components that have two states: on or off, corresponding directly to the binary values 1 and 0.

At the heart of these digital circuits are **transistors**. A transistor is a tiny electronic device (typically less than 45 nanometres in size) that allows current to flow through it, functioning as a switch. Computers are made up of billions of connected transistors.

Figure 4.3: A Transistor.

These transistors can be wired together in such a way that they store information in binary form — each 0 or 1 represents a single bit of information. Furthermore, they can be used to perform logical operations on these bits.

Every computation or decision made by a computer involves a series of logical operations — such as AND, OR, and NOT — that are described and optimised using Boolean expressions. These logical operations are fundamental to controlling the flow of data and decision-making in software and hardware systems. For example:

- In **processors**, Boolean algebra is used to create **logic gates** and circuits that handle data processing, arithmetic operations, and decision-making.
- In **software**, conditional statements like IF and ELSE are based on Boolean logic, determining which paths a program will follow.

Beyond digital circuits, Boolean algebra also underpins essential concepts in **logic programming**, **database querying**, and **search algorithms**, where conditions must be evaluated as TRUE or FALSE.

At the physical level, the circuits in digital devices operate by processing binary inputs to produce a binary output. Each of these circuits can be modelled using Boolean functions, which are expressions that describe how input values are manipulated to produce an output. For example, a simple AND gate outputs 1 only if both of its inputs are 1, which aligns directly with the Boolean operation $x \wedge y$.

Boolean algebra is not only a way of simplifying logical reasoning but also a tool for optimising digital circuit designs. By simplifying Boolean expressions, designers can reduce the number of logic gates required, which leads to more efficient circuits that use less power and occupy less space on a chip.

4.2 Basic Operations on Bits

Boolean operators define the basic rules for manipulating binary values (0 and 1). The three primary operations in Boolean algebra are:

Primary Boolean Operations

- **Complementation** (denoted as `NOT`, e.g. $\overline{0}$)
- **Boolean Sum** (denoted as `OR`, e.g. $1 + 0$)
- **Boolean Product** (denoted as `AND`, e.g. $1 \cdot 0$)

The **complement** of a Boolean value is the inverse of its state. It is denoted with a bar over the variable. For instance:

$$\overline{0} = 1, \quad \overline{1} = 0$$

The complement operation flips the value: if the input is 0, the output is 1, and vice versa.

The Boolean sum, or `OR`, is a logical operation that returns 1 if at least one of the operands is 1. Its results are as follows:

$$1 + 1 = 1, \quad 1 + 0 = 1, \quad 0 + 1 = 1, \quad 0 + 0 = 0$$

This operation is analogous to addition but follows different rules for binary logic.

The Boolean product, or `AND`, returns 1 only if both operands are 1. Its results are:

$$1 \cdot 1 = 1, \quad 1 \cdot 0 = 0, \quad 0 \cdot 1 = 0, \quad 0 \cdot 0 = 0$$

In some cases, the multiplication symbol (`AND`) can be omitted for clarity, similar to how it is omitted in algebraic multiplication.

We can summarise the considerations about the basic Boolean operators in the following table:

`OR`	`AND`	`NOT`
$0 + 0 = 0$	$0 \cdot 0 = 0$	$\overline{0} = 1$
$0 + 1 = 1$	$0 \cdot 1 = 0$	$\overline{1} = 0$
$1 + 0 = 1$	$1 \cdot 0 = 0$	
$1 + 1 = 1$	$1 \cdot 1 = 1$	

Table 4.1: Basic Boolean Operators.

Remark: A **Boolean variable** can only take on two possible values: 0 (`FALSE`) or 1 (`TRUE`). These variables represent the states in logic circuits. For example, given $x = 1$ and $y = 0$, you can apply Boolean operations:

$$x + y = 1, \quad x \cdot y = 0, \quad \overline{x} = 0$$

We will typically use x, y and z as our Boolean variables.

Operator Precedence

When evaluating expressions with multiple Boolean operators, the following order of operations is applied:

1. Complements (NOT) are evaluated first.
2. Boolean products (AND) are computed next.
3. Boolean sums (OR) are evaluated last.

So, for example, $x \cdot y + z$ means $(x \cdot y) + z$. Most of the time, however, we will write the parenthesis to avoid misunderstandings. Also note that parentheses can be used to override the default precedence, ensuring that certain operations are evaluated before others.

Consider the expression:

$$\overline{x} \cdot y + z$$

According to the precedence rules, the complement (NOT) of x is evaluated first, followed by the Boolean product (AND) between $\overline{x}$ and y, and finally, the Boolean sum (OR) with z. This can be written explicitly as:

$$(\overline{x} \cdot y) + z$$

Now consider another expression:

$$\overline{x \cdot (y + z)}$$

In this case, the parentheses override the default precedence. The Boolean sum (OR) between y and z is evaluated first, followed by the Boolean product (AND) with x, and finally, the complement (NOT) of the entire result.

Example 4.1 Find the value of $1 \cdot 0 + \overline{(0 + 1)}$.

Solution: Using the definitions of complementation, sum, product and the order of operations, it follows that

$$\begin{aligned} 1 \cdot 0 + \overline{(0 + 1)} &= 0 + \overline{1} \\ &= 0 + 0 \\ &= 0 \end{aligned}$$

◀

It turns out that Boolean operators are intimately linked with logical operators.

Logical Operators

The complement, Boolean sum, and Boolean product correspond to the logical operators $\neg$, $\vee$, and $\wedge$, respectively. In this context, 0 corresponds to FALSE ($\mathbf{F}$) and 1 corresponds to TRUE ($\mathbf{T}$).

Equalities in Boolean algebra can be directly translated into equivalences of compound propositions in logic. Similarly, logical equivalences can be translated into equalities in Boolean algebra. These translations allow us to move between Boolean algebra and propositional logic seamlessly.

Example 4.2 Translate $1 \cdot 0 + \overline{(0 + 1)} = 0$, the equality found in Example 4.1, into a logical equivalence.

Solution: $(\mathbf{T} \wedge \mathbf{F}) \vee \neg(\mathbf{T} \vee \mathbf{F}) \equiv \mathbf{F}$ ◀

Since logic is a whole field in its own right, we will not consider this any further but simply note that the *equivalence* is uncanny (no pun intended).[1]

4.3 Boolean Functions

A **Boolean function** maps Boolean inputs to a Boolean output. Consider the Boolean function $F(x, y, z)$:

$$F(x, y, z) = (x \cdot \overline{y}) + z$$

This function returns the result of the Boolean `OR` between two terms: $x \cdot \overline{y}$ (the `AND` of x and the complement of y) and z. In the following, the function is displayed in a table, which shows the value of F for all combinations of values of the inputs:

x	y	z	$F(x, y, z)$
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	0
1	1	1	1

Table 4.2: Truth Table for $F(x, y, z) = (x \cdot \overline{y}) + z$.

Example 4.3 Consider the Boolean function defined as:

$$F(x, y, z) = xy + \overline{z}$$

Display the truth table for this function.

Solution: This function is displayed in the table below, which shows the value of F for all combinations of the input values x, y, and z:

The columns xy and $\overline{z}$ are "helping" columns that assist in the calculation of F, since they are not strictly necessary. They break down the expression $F(x, y, z)$ into smaller parts to make it easier to compute the final result.

◀

[1]Notice that we used '≡' in the previous example. The equivalence symbol is more specific in logic (and Boolean algebra), signifying logical equivalence. Two expressions are logically equivalent if they always yield the same truth value, regardless of the values of the variables involved. Using '≡' emphasises that the two expressions are not just equal for specific values, but that they are structurally identical in terms of logic. We choose to use '=' because it communicates that the expressions behave identically on all inputs.

x	y	z	xy	$\overline{z}$	$F(x,y,z)$
0	0	0	0	1	1
0	0	1	0	0	0
0	1	0	0	1	1
0	1	1	0	0	0
1	0	0	0	1	1
1	0	1	0	0	0
1	1	0	1	1	1
1	1	1	1	0	1

Table 4.3: Truth table for the Boolean function $F(x,y,z) = xy + \overline{z}$.

Simplifying a Boolean Expression

The table below shows the output for the functions $F(x,y) = (x\overline{y} + xy) \cdot y$ and $G(x,y) = xy$ for all values of x and y:

x	y	xy	$\overline{y}$	$x\overline{y}$	$xy + x\overline{y}$	$F(x,y)$	$G(x,y)$
0	0	0	1	0	0	0	0
0	1	0	0	0	0	0	0
1	0	0	1	1	1	0	0
1	1	1	0	0	1	1	1

Table 4.4: Comparison of $F(x,y) = (x\overline{y} + xy) \cdot y$ and $G(x,y) = xy$.

These functions produce the same output for all values of x and y. That is, these two functions are equivalent!

Boolean expressions can often be simplified significantly. In the following section, we will delve into some rules and techniques to achieve this simplification.

Boolean Identities

Boolean identities are fundamental rules that define the relationships and operations between Boolean variables. These identities allow us to simplify and manipulate Boolean expressions. By applying these identities, complex Boolean expressions can be reduced to simpler forms, making them easier to evaluate or implement in digital circuits. Whether using commutative, associative, distributive, or De Morgan's laws, these identities provide the building blocks for efficient logic manipulation, helping us understand the behaviour of Boolean functions at a deeper level.

We start out by inspecting the must fundamental of these identities:

Fundamental Boolean Properties

- **Double Complement:**
  $\overline{\overline{x}} = x$
- **Idempotent Laws:**
  $x + x = x, \quad x \cdot x = x$
- **Identity Laws:**
  $x + 0 = x, \quad x \cdot 1 = x$
- **Domination Laws:**
  $x + 1 = 1, \quad x \cdot 0 = 0$
- **Unit Property:**
  $\overline{x} + x = 1$
- **Zero Property:**
  $\overline{x} \cdot x = 0$

These identities represent the most fundamental properties of Boolean algebra, dealing with the behavior of `OR`, `AND`, and the identity elements (0 and 1).

Example 4.4 Show that $(1 \cdot 1) + (\overline{0 \cdot 1} + 0) = 1$.

Solution: To show that $(1 \cdot 1) + (\overline{0 \cdot 1} + 0) = 1$, let's break down and simplify the expression step by step using Boolean algebra identities.

- **Step 1:** Simplify the first part $(1 \cdot 1)$ Using the Idempotent Law $(x \cdot x = x)$ or simply calculating the result:
  $1 \cdot 1 = 1$
  So, the expression becomes:
  $1 + (\overline{0 \cdot 1} + 0)$
- **Step 2:** Simplify the second part $(\overline{0 \cdot 1} + 0)$ First, simplify $0 \cdot 1$ using the Domination Law $(0 \cdot x = 0)$:
  $0 \cdot 1 = 0$
  So, the expression becomes:
  $1 + (\overline{0} + 0)$
- **Step 3:** Simplify the complement $\overline{0}$ Using the Complement Law $(\overline{0} = 1)$:
  $\overline{0} = 1$
  Thus, the expression becomes:
  $1 + (1 + 0)$
- **Step 4:** Simplify the inner expression $(1 + 0)$
  Using the Identity Law $(x + 0 = x)$:
  $1 + 0 = 1$
  So, the expression now becomes:
  $1 + 1$
- **Step 5:** Simplify $1 + 1$
  Using the Idempotent Law $(x + x = x)$ or the Domination Law:
  $1 + 1 = 1$
  The expression is true.

Thus, we have shown that:

$$(1\cdot 1)+(\overline{0\cdot 1}+0)=1$$

◀

Next we take a look at some other important laws:

Structural Boolean Laws

- **Commutative Laws:**
  $x+y=y+x, \quad x\cdot y=y\cdot x$
- **Absorption Laws:**
  $x+(x\cdot y)=x, \quad x\cdot(x+y)=x, \quad \bar{x}\cdot\bar{y}+x=\bar{y}+x$
- **De Morgan's Laws:**
  $\overline{x\cdot y}=\overline{x}+\overline{y}, \quad \overline{x+y}=\overline{x}\cdot\overline{y}$
- **Associative Laws:**
  $x+(y+z)=(x+y)+z, \quad x\cdot(y\cdot z)=(x\cdot y)\cdot z$
- **Distributive Law:**
  $x\cdot(y+z)=(x\cdot y)+(x\cdot z), \quad x+(y\cdot z)=(x+y)\cdot(x+z)$

Together with the basic identities, the structural laws allow us to reduce more complex expressions.

Example 4.5 Show that $(x\cdot y)+\overline{(x+z)\cdot\overline{y}}=y+\overline{x}\cdot\overline{z}$

Solution:

Note that we do not explicitly state the use of the **Commutative Laws**.

- **Step 1:** Apply **De Morgan's Law** to the complement
  $\overline{(x+z)\cdot\overline{y}}$
  The expression becomes:
  $(x\cdot y)+\overline{(x+z)}+\overline{\overline{y}}$
- **Step 2:** Use **Double Complement Law**
  $\overline{\overline{y}}$
  The expression becomes:
  $(x\cdot y)+\overline{(x+z)}+y$
- **Step 3:** Apply **De Morgan's Law** to the complement
  $\overline{(x+z)}$
  The expression becomes:
  $(x\cdot y)+\overline{x}\cdot\overline{z}+y$
- **Step 4:** Apply the **Distributive Law** in reverse
  $(x\cdot y)+\overline{x}\cdot\overline{z}+y=y(x+1)+\overline{x}\cdot\overline{z}$
- **Step 5:** Use the **Domination Law** to simplify $x+1$:
  So, the expression simplifies to:
  $y+\overline{x}\cdot\overline{z}$

Thus, we have shown that:

$$(x \cdot y) + \overline{(x+z) \cdot \overline{y}} = y + \overline{x} \cdot \overline{z}$$

In addition to the basic operators, we have other commonly used operators in Boolean algebra:

- NOT AND (NAND): This is the complement of the AND operation. It returns true unless both inputs are true. In Boolean algebra, NAND is denoted by the symbol $\uparrow$ and can be written as:
  $x \uparrow y = \overline{x \cdot y}$
- NOT OR (NOR): The NOR operator is the complement of the OR operation. It returns true only when both inputs are false. Denoted by the symbol $\downarrow$, it can be expressed as:
  $x \downarrow y = \overline{x + y}$
- Exclusive OR (XOR): The XOR operator returns true if exactly one of the inputs is true, but not both. This is particularly useful in arithmetic operations such as binary addition. Denoted by $\oplus$, the Boolean expression for XOR is:
  $x \oplus y = (x \cdot \overline{y}) + (\overline{x} \cdot y)$

We summarise the additional operators in the following table:

NAND	NOR	XOR
$0 \uparrow 0 = 1$	$0 \downarrow 0 = 1$	$0 \oplus 0 = 0$
$0 \uparrow 1 = 1$	$0 \downarrow 1 = 0$	$0 \oplus 1 = 1$
$1 \uparrow 0 = 1$	$1 \downarrow 0 = 0$	$1 \oplus 0 = 1$
$1 \uparrow 1 = 0$	$1 \downarrow 1 = 0$	$1 \oplus 1 = 0$

Table 4.5: Additional Boolean Operators.

Here are all the operators in one table:

OR	AND	NOT	NAND	NOR	XOR
$0+0=0$	$0 \cdot 0 = 0$	$\overline{0} = 1$	$0 \uparrow 0 = 1$	$0 \downarrow 0 = 1$	$0 \oplus 0 = 0$
$0+1=1$	$0 \cdot 1 = 0$	$\overline{1} = 0$	$0 \uparrow 1 = 1$	$0 \downarrow 1 = 0$	$0 \oplus 1 = 1$
$1+0=1$	$1 \cdot 0 = 0$		$1 \uparrow 0 = 1$	$1 \downarrow 0 = 0$	$1 \oplus 0 = 1$
$1+1=1$	$1 \cdot 1 = 1$		$1 \uparrow 1 = 0$	$1 \downarrow 1 = 0$	$1 \oplus 1 = 0$

Table 4.6: Basic and Additional Boolean Operators.

Now we can state the final, and more advanced, Boolean identities:

Advanced Boolean Identities

- **Exclusive OR (XOR):**
  $x \oplus y = (x \cdot \overline{y}) + (\overline{x} \cdot y)$
- **Exclusive NOR (XNOR):**
  $\overline{x \oplus y} = (x \cdot y) + (\overline{x} \cdot \overline{y})$
- **Disappearing Opposite:**
  $x + \overline{x} \cdot y = x + y$
- **FOIL Law (First, Outer, Inner, Last):**
  $(x + y) \cdot (z + w) = (x \cdot z) + (x \cdot w) + (y \cdot z) + (y \cdot w)$

4.4 Logic Gates

Logic gates, the fundamental building blocks of digital circuits, manage the flow of electric current entering and leaving circuits. By receiving binary inputs and generating outputs, logic gates regulate electrical signals. These gates operate within transistors or as parts of larger systems like arithmetic units, multiplexers, and microprocessors, enabling essential electronic logic operations.

Types of Logic Gates

There are various examples of logic gates that function differently depending on the type of binary input they receive. In Boolean algebra, these gates correspond to basic logical operations, such as AND, OR, and NOT. Below are the most common types of logic gates found in most electronic circuits:

- AND
- OR
- NOT
- NOR
- XOR
- NAND

The activation of each of these logic gates depends on the presence or absence of a specific binary input that the gate is designed to recognise. If the input matches its configuration, the logic gate will output a 1; if it does not match, the output will be a 0.

AND Logic Gates

An AND gate functions as an all-or-nothing device. To activate the gate, all inputs must be 1. If even one input is 0, the output will be 0.

OR Logic Gates

OR gates are less restrictive than AND gates. If at least one input is 1, the output will be 1, regardless of the remaining inputs.

NOT Logic Gates

A NOT gate inverts its input. It has a single input and a single output, where the output is always the opposite of the input. For example, if the input is 1, the output will be 0, and vice versa.

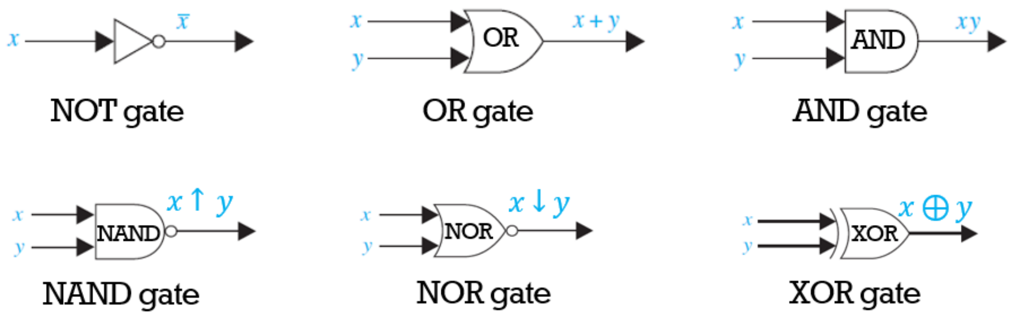


Figure 4.4: ANSI Symbols of Most Common Logic Gates. (Rosen, 2012).

NOR Logic Gates

A NOR gate combines the characteristics of an OR gate with an inverter. The output of a NOR gate is 1 only when all inputs are 0. Any other input combination results in a 0 output.

XOR Logic Gates

An XOR (Exclusive OR) gate outputs a 1 only when exactly one of its inputs is 1. If both inputs are the same, either both 0 or both 1, the output will be 0. It functions like an OR gate, but with the additional condition that both inputs cannot be 1 simultaneously to produce a 1.

NAND Logic Gates

A NAND gate is a combination of an AND gate and a NOT gate. The output will be 0 only if all inputs are 1. In any other situation, the output will be 1.

Similarly to how we previously reduced Boolean expressions to simpler forms, we can achieve the same outputs using different circuit designs as seen in figure 4.5.

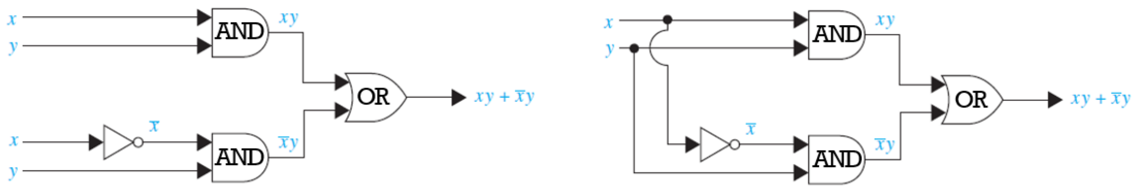


Figure 4.5: Two Equivalent Logic Gates. (Rosen, 2012).

We conclude this chapter by exploring the idea of **functional completeness**, also known as a **universal gate**. This concept refers to a set of logic gates that can be combined to implement any Boolean function. In essence, a functionally complete set of gates, like NAND or NOR, can replicate the behaviour of all other basic logic gates, making them versatile tools in digital circuit design.

Functional Completeness and a Universal Gate

An operator or set of operators is considered **functionally complete** if it can be used to express any Boolean function. Both NAND and NOR gates are functionally complete, or **universal gates**, meaning that any logical expression can be constructed using only NAND or only NOR.

The reason the NAND (or the NOR) gate is functionally complete lies in its ability to mimic other Boolean operations. Using just NAND operations, we can construct the fundamental operations of NOT, AND, and OR, which are the building blocks for any Boolean function.

- NOT ($\overline{x}$): A NOT gate can be constructed by connecting both inputs of a NAND gate to the same variable: $\overline{x} = x \uparrow x$
- AND ($x \cdot y$): The AND operation can be achieved by first applying a NAND gate and then passing the result through a NOT (which, as seen, can also be constructed using NAND):
  $x \cdot y = \overline{x \uparrow y} = (x \uparrow y) \uparrow (x \uparrow y)$
- OR ($x + y$): The OR operation can be created using multiple NAND gates, taking advantage of De Morgan's laws: $x + y = (x \uparrow x) \uparrow (y \uparrow y)$

The table below demonstrates how we can construct the basic Boolean operations using only NAND gates. Notice how the different columns represent the transformations of variables x and y using combinations of NAND gates to form the logical operations NOT, AND, and OR:

Input	**Input**	**NOT**	**AND**	**OR**
x	y	$x \uparrow x$	$(x \uparrow y) \uparrow (x \uparrow y)$	$(x \uparrow x) \uparrow (y \uparrow y)$
0	0	1	0	0
0	1	1	0	1
1	0	0	0	1
1	1	0	1	1

Table 4.7: Constructing Basic Boolean Operators with NAND Gates.

Since we have seen how to build all Boolean functions using just NAND gates, we now understand that all logic operations in a computer can be implemented using this single type of gate. This makes the NAND gate incredibly important in digital circuit design, as it allows for simpler and more efficient hardware implementation. In modern computer systems, transistors are used to build NAND gates, meaning that any logical operation can ultimately be represented by a combination of transistors.

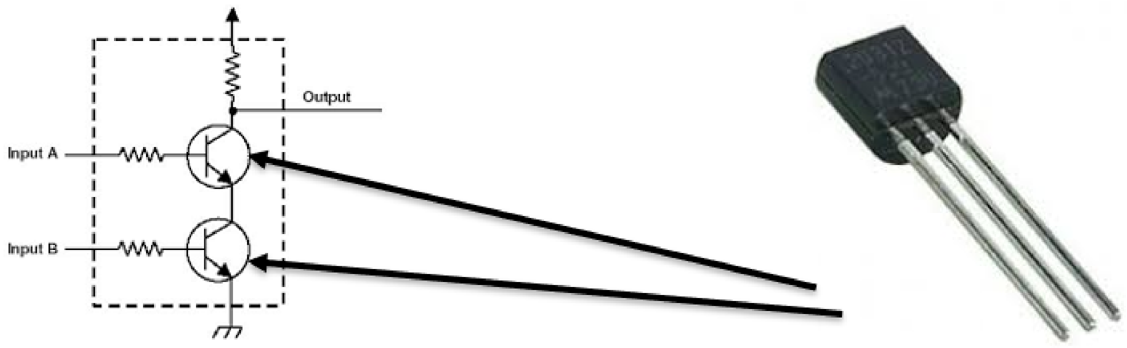


Figure 4.6: Logic Gates in Computers.

Thus, the concept of functional completeness is not just theoretical; it is a powerful principle that allows computer engineers to design flexible and efficient circuits using minimal components.

Chapter 5 Combinatorics and Probability Theory

Imagine you are tasked with forming teams of 3 for a semester project in a class of 45 students. Initially, the order in which you choose the team members does not matter, so you are just concerned with combinations. The number of ways to form a team of 3 from 45 students comes out to 14,190 possibilities!

The following semester introduces the students to Scrum project management, where each team must have three specific roles: Scrum Master, Product Owner, and Development Team. This small change, specifying roles, suddenly transforms the problem from a simple *combination* into a *permutation*. Now, the number of possible ways to assign these roles leaps to 85,140!

Frustrated by the sheer number of options, the 45 students throw a party to relax. Being well-mannered, they decide that everyone should shake hands with every other person exactly once. After a few minutes, they calculate the total number of handshakes — 990. The students are once again surprised by how something as simple as shaking hands can add up so quickly.

As the night progresses, one student proposes a fun game — a random drawing for five door prizes, each unique. With 45 students in attendance and only five prizes available, the chance of winning nothing becomes a concerning 89 per cent. The students quickly realize that the odds are not in their favor.

Not ready to give up on their luck, a smaller group decides to flip a coin 10 times, with the hope of landing exactly five heads to win the game. However, when they learn that the probability of this happening is only about 25 per cent, their spirits dampen further.

The students conclude that rather than relying on chance, it is time to dive deeper into understanding combinatorics and probability theory. Armed with this knowledge, they can better predict outcomes and avoid future disappointments at both parties and project planning.

5.1 Sample Space and Events

A **random experiment** is one that can lead to different outcomes, even when repeated under the same conditions. This randomness is a fundamental aspect of many engineering tasks.

Think of it this way: Let us say you are testing the speed of a website under different conditions. Sometimes it loads quickly, and other times it is slower. Even if you are using the same code and server, things like network traffic or server load make the results vary each time.

Definition 5.1 (Random Experiment)

A random experiment is one that can give different results, even if you do everything the same each time. ♣

Or, imagine you are measuring the signal strength in a wireless device. You might get slightly different readings each time because of things like interference or small changes in the environment.

This randomness shows up all over the place, from software performance tests to electrical engineering experiments. It is important to expect it and include it in your thinking. Otherwise, you might make decisions based on incomplete or misleading data. When you account for random variation, you can make smarter predictions and designs.

To model and analyze a random experiment, it is crucial to understand the set of possible outcomes that can occur. In probability theory, this set is called the **sample space**, denoted by S. A sample space can be either **discrete** (consisting of a finite or countably infinite set of outcomes) or **continuous** (containing an interval of real numbers). The exact definition of a sample space often depends on the objectives of the analysis.

An **outcome** is a single possible result of the random experiment, and an **event** is any subset of the sample space, which may consist of one or more outcomes. Below are some examples to illustrate these concepts:

Example 5.1 Network Latency

Consider an experiment where you measure the latency of data packets in a network. The sample space can be defined based on the type of measurements:

- If latency is measured as a positive real number, the sample space is continuous:
  $S = \{x \mid x > 0\}$.
- If it is known that latency ranges between 10 and 100 milliseconds, the sample space can be refined to:
  $S = \{x \mid 10 \leq x \leq 100\}$.
- If the objective is to categorize latency as low, medium, or high, the sample space becomes discrete:
  $S = \{\text{low}, \text{medium}, \text{high}\}$.
- For a simple evaluation of whether the latency meets a standard threshold, the sample space can be reduced to:
  $S = \{\text{pass}, \text{fail}\}$.

Each outcome in these sample spaces represents a single possible latency measurement, and events can be defined as sets of outcomes, such as "latency is high."

Example 5.2 Software Release Testing

Imagine a software testing process where each test case can either pass or fail. The sample space for a single test case is discrete and can be represented as:

$$S = \{\text{pass}, \text{fail}\}.$$

If you run three test cases, the combined sample space for all possible outcomes is:

$$\begin{aligned} S = \{&(\text{pass, pass, pass}), (\text{pass, pass, fail}), (\text{pass, fail, pass}),\\ &(\text{pass, fail, fail}), (\text{fail, pass, pass}), (\text{fail, pass, fail}),\\ &(\text{fail, fail, pass}), (\text{fail, fail, fail})\}. \end{aligned}$$

This sample space includes all sequences of outcomes for the three tests.

- The total number of possible outcomes is $2^3 = 8$.
- An event could be defined as "at least one test fails," which would include outcomes like (fail, pass, pass), (pass, fail, fail), and others where at least one test fails.

Example 5.3 Component Quality in Manufacturing

A company manufactures electronic components, and each component is tested for compliance with quality standards. The test can return one of three outcomes: `pass`, `marginal`, or `fail`. The sample space is:

$$S = \{\text{pass}, \text{marginal}, \text{fail}\}.$$

Event: Suppose we are interested in the event that a component does not pass the quality test. This event is a set of outcomes:

$$E = \{\text{marginal}, \text{fail}\}.$$

Understanding the nature of sample spaces, outcomes, and events is fundamental in probability, as it allows us to define and work with probabilities of complex scenarios in various engineering contexts. Let us summarise these key concepts:

Definition 5.2 (Sample Spaces, Outcomes, and Events)

Sample Space: The set of all possible outcomes of a random experiment is called the sample space, denoted by S. Outcomes can be discrete or continuous, depending on the nature of the experiment.
Outcome: A single possible result of a random experiment.
Event: Any subset of the sample space, which may consist of one or more outcomes. ♣

5.2 Types of Events in Probability Theory

Events in probability theory can be classified based on their relationships with each other and the sample space. The study of probability and Boolean algebra often intersect (no pun intended), as both involve operations on sets and logical reasoning. In probability, we work with events, which are subsets of a sample space, while Boolean algebra uses logical statements that can be true or false. Despite their different applications, the operations on events correspond directly to Boolean algebra operators. Here are some common types of events and their relationships to Boolean operations.

Union and Logical `OR`

The **union** of two events A and B represents the event that at least one of the events occurs. Mathematically, it includes all outcomes that are in A, B, or both. For example, if A is the event of getting a head when flipping a coin, and B is the event of getting a tail, then $A \cup B = \{\text{heads, tails}\}$. This is analogous to the logical `OR` operator ($\vee$) in Boolean algebra, where the statement $A + B$ is true if at least one of A or B is true.

Probability: $A \cup B = \{\text{outcomes in } A \text{ or } B \text{ or both}\}$

Boolean Algebra: $A + B = \text{true if } A \text{ is true or } B \text{ is true}$

Intersection and Logical `AND`

The **intersection** of two events A and B represents the event that both A and B occur simultaneously. It includes only the outcomes that are in both A and B. For example, if A is the event of getting an even number on a die roll, and B is the event of getting a number greater than 3, then $A \cap B = \{4, 6\}$. This corresponds to the logical `AND` operator ($\cdot$) in Boolean algebra, where $A \wedge B$ is true only if both A and B

are true.

Probability: $A \cap B = \{\text{outcomes in both } A \text{ and } B\}$

Boolean Algebra: $A \cdot B = \text{true only if both } A \text{ and } B \text{ are true}$

Complement and Logical `NOT`

The complement of an event A represents all outcomes in the sample space that are not in A. If A is the event of rolling a 4 on a die, then the complement $\overline{A}$ is the event of not rolling a 4, i.e., $\overline{A} = \{1, 2, 3, 5, 6\}$. This operation is analogous to the logical NOT operator $(\overline{A})$ in Boolean algebra, where $\overline{A}$ is true if A is false.

Probability: $A^c = \{\text{outcomes not in } A\}$

Boolean Algebra: $\overline{A} = \text{true only if } A \text{ is false}$

Remark: In the context of set theory and probability theory, the notation of $\overline{A}$ is commonly used along with A^c to denote the complement of event A and even A' in some cases.

Additional Events

In addition to the basic operations of `OR`, `AND`, and `NOT`, there are several other ways to combine and analyze events in probability and Boolean algebra. These operations help refine our understanding of event relationships and provide a framework for solving complex problems. Some of these operations include:

- **Difference ($A - B$)**: The difference between two events A and B (sometimes denoted as $A \setminus B$) is the event that occurs if A happens but B does not. For example, if A is the event of drawing a red card from a deck, and B is the event of drawing a heart, then $A - B$ is the event of drawing a red card that is not a heart (i.e., a diamond).
- **Symmetric Difference ($A \triangle B$)**: The symmetric difference between two sets (or events) A and B consists of elements that are in either A or B but not in their intersection. This operation is analogous to the logical `XOR` operator $(A \oplus B)$ in Boolean algebra, where $A \oplus B$ is true only if exactly one of A or B is true, but not both.

  Probability: $A \triangle B = \{\text{outcomes in either } A \text{ or } B \text{ but not both}\}$

  Boolean Algebra: $A \oplus B = \text{true only if exactly one of } A \text{ or } B \text{ is true}$
- **Mutually Exclusive Events (or Disjoint)**: Two events are mutually exclusive (or disjoint) if they cannot occur at the same time. Their intersection is empty: $A \cap B = \emptyset$. For example, the events of rolling a 2 and rolling a 4 on a die are mutually exclusive because they cannot happen simultaneously.
- **Collectively Exhaustive Events**: A set of events is collectively exhaustive if at least one of the events must occur. Together, they cover the entire sample space. For example, when rolling a die, the events $A = \{\text{even numbers}\}$ and $B = \{\text{odd numbers}\}$ are collectively exhaustive because every outcome is either even or odd.

Remark: In the context of set theory and probability theory, the notation of $A \oplus B$ is commonly used.

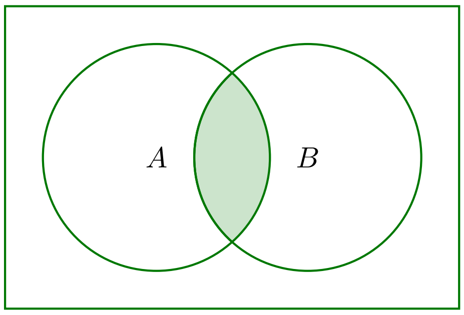
We can summarise the considerations of this section in the following table:

Operation	Boolean Algebra	Logic	Set Theory
NOT	$\overline{x}$	$\neg x$	A^c or $\overline{A}$ or A'
OR	$+$	$\vee$	$\cup$
AND	$\cdot$	$\wedge$	$\cap$
NAND	$\overline{x \cdot y}$	$\neg(x \wedge y)$	$(A \cap B)^c$ or $\overline{A \cap B}$
NOR	$\overline{x + y}$	$\neg(x \vee y)$	$(A \cup B)^c$ or $\overline{A \cup B}$
XOR (Symmetric Difference)	$x \oplus y$	$(x \wedge \neg y) \vee (\neg x \wedge y)$	$A \triangle B$ or $A \oplus B$
Difference	$x \cdot \overline{y}$	$x \wedge \neg y$	$A - B$ or $A \setminus B$

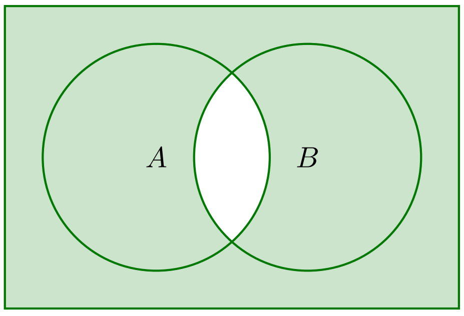
Table 5.1: Comparison of Operators in Boolean Algebra, Logic, and Set Theory

Venn diagrams are a useful tool for visualizing the relationships between sets, and they are also commonly used to illustrate relationships between events. By using Venn diagrams, we can visually represent a sample space and the events within it.

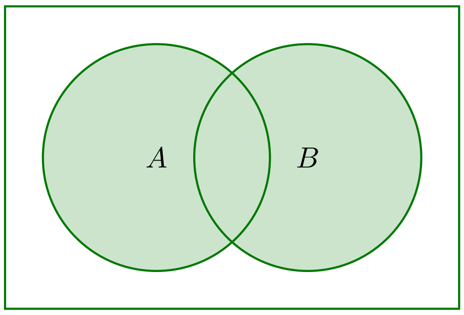
Figure 5.1a shows the intersection, representing the outcomes common to both events A and B. Figure 5.1b represents the outcomes that are not in the intersection of A and B. Figure 5.1c depicts the union, showing all outcomes that are in either event A, event B, or both. Figure 5.1d represents the outcomes that are in event A but not in event B, which is the difference between A and B. Figure 5.1e shows all the elements that are not A. Figure 5.1g illustrates mutually exclusive (or *disjoint*) events, where the two sets do not overlap, indicating that events A and B have no common outcomes.



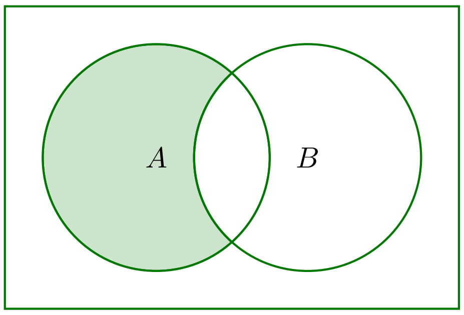
(a) $A \cap B$



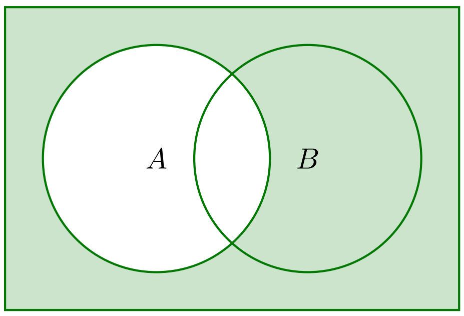
(b) $\overline{A \cap B}$



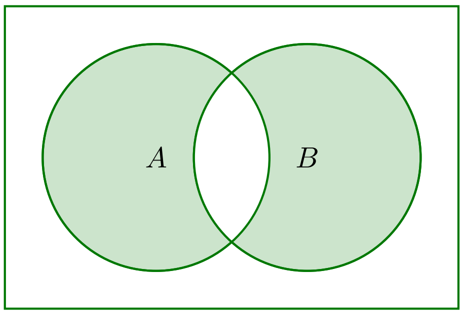
(c) $A \cup B$



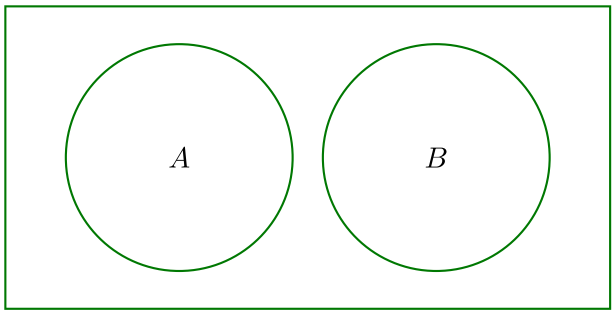
(d) $A - B$



(e) $\overline{A}$



(f) $A \triangle B$ (`XOR`)



(g) Mutually Exclusive (Disjoint)

Figure 5.1: Venn Diagrams Illustrating Various Set Operations

5.3 Counting Principles

In many problems across mathematics, computer science, and engineering, determining the number of ways certain events can occur is important. Whether you're arranging elements, selecting groups, or navigating through complex scenarios, counting techniques provide the foundational tools to solve these problems. These techniques go beyond simple arithmetic and allow us to tackle questions like:

- How many ways can we arrange a set of objects?
- In how many different paths can a process unfold?
- What is the probability of a specific event occurring given multiple possibilities?

Counting techniques, such as permutations, combinations, and the multiplication rule, help us quantify these possibilities systematically.

Multiplication Rule

We start out by discussing the most basic counting principle: the **multiplication rule**:

Theorem 5.1 (Multiplication Rule)

Let an operation be described as a sequence of k steps. Assume the following conditions:

- There are n_1 ways to complete step 1.
- There are n_2 ways to complete step 2 for each way of completing step 1.
- There are n_3 ways to complete step 3 for each way of completing step 2, and so on.

Then, the total number of ways to complete the entire operation is given by:

$$n_1 \times n_2 \times \cdots \times n_k.$$

♡

Example 5.4 Suppose you are choosing a meal at a restaurant. You have the following options:

- 3 choices for the main course.
- 4 choices for the side dish.
- 2 choices for the drink.

Using the multiplication rule, the total number of ways to choose a meal is:

$$3 \times 4 \times 2 = 24.$$

Therefore, there are 24 different meal combinations available.

Example 5.5 Automobile Options

An automobile manufacturer provides vehicles equipped with selected options. Each vehicle is ordered

- With or without an automatic transmission
- With or without a sunroof
- With one of three choices of a stereo system
- With one of four exterior colors

If the sample space consists of the set of all possible vehicle types, what is the number of outcomes in the sample space?

Solution: Using the multiplication rule, we can calculate the total number of possible vehicle types by multiplying the number of choices for each option:

- 2 choices for the transmission (with or without automatic transmission)
- 2 choices for the sunroof (with or without sunroof)
- 3 choices for the stereo system
- 4 choices for the exterior color

Therefore, the total number of possible vehicle types is:

$$2 \times 2 \times 3 \times 4 = 48$$

So, there are 48 different possible vehicle types in the sample space. ◀

Replacement and Order in Counting

Next we turn to an important distinction between with and without replacement in counting principles:

Definition 5.3 (Counting with and without Replacement)

When counting the number of ways to select objects from a set, two common scenarios are:

- **With Replacement**: An object can be selected more than once.
- **Without Replacement**: Once an object is selected, it cannot be chosen again.

♣

Example 5.6 Suppose you have a bag containing 5 different colored balls. You draw 2 balls:

- **With Replacement**: The first ball is placed back in the bag before drawing the second. There are $5 \times 5 = 25$ possible outcomes.
- **Without Replacement**: The first ball is not placed back, so the number of outcomes is $5 \times 4 = 20$.

Example 5.7 Three people are drawing cards one after another from a standard deck of 52 cards. The goal is to find the Ace of Spades. Let's examine the two scenarios: with replacement and without replacement.

Without Replacement

In this scenario, each card drawn is not put back into the deck, reducing the total number of cards available after each draw.

- First Draw: The first person has a $\frac{1}{52}$ chance of drawing the Ace of Spades.
- Second Draw: If the first person does not draw the Ace of Spades, there are now 51 cards left, and the second person has a $\frac{1}{51}$ chance of drawing the Ace of Spades.
- Third Draw: If the Ace of Spades has not been drawn by the first two people, the third person has a $\frac{1}{50}$ chance of drawing it.

The probabilities change with each draw because the total number of cards decreases, and previously drawn cards are not available.

With Replacement

In this scenario, each card drawn is returned to the deck and reshuffled before the next person draws. This keeps the total number of cards constant.

- First Draw: The first person has a $\frac{1}{52}$ chance of drawing the Ace of Spades.
- Second Draw: Since the card is replaced and shuffled back into the deck, the second person also has a $\frac{1}{52}$ chance of drawing the Ace of Spades.
- Third Draw: Similarly, the third person has a $\frac{1}{52}$ chance of drawing the Ace of Spades.

The probabilities remain the same for each draw because the deck is reset to its original state after each draw.

Example 5.8 Imagine you have a group of 5 students: Alice, Bob, Charlie, David, and Eve. You need to select 2 of them for different scenarios, illustrating when the order of selection matters and when it does not.

- **Order Matters**: Selecting Alice as Captain and Bob as Assistant Captain is a different outcome than selecting Bob as Captain and Alice as Assistant Captain.

Now, imagine you are simply selecting 2 students to form a study group with no specific roles assigned. Here, the order does not matter.

- **Order does not matters:** Choosing Alice and Bob is considered the same outcome as choosing Bob and Alice; there is no distinction between the two orders since there are no assigned roles.

Example 5.8 illustrates the distinction between *permutations* and *combinations*, two fundamental counting principles that are widely used in probability theory and combinatorics.

Definition 5.4 (Permutation and Combination)

- **Permutation (Order Matters)**: Different sequences are counted as distinct outcomes, leading to a higher count.
- **Combination (Order Does Not Matter)**: Sequences are treated as identical, resulting in a lower count.

♣

We will first discuss permutations, which are used when the order of selection matters.

Permutations

Consider a set of elements, such as $S = \{a, b, c\}$. A permutation of the elements is an ordered sequence of the elements. For example, abc, acb, bac, bca, cab, and cba are all of the permutations of the elements of S.

Definition 5.5 (Permutations of n Objects)

The permutation of n objects, i.e. an ordered arrangement of n objects, is $n!$ (read as "n factorial"), where:

$$n! = n \times (n-1) \times (n-2) \times \cdots \times 1$$

♣

This outcome is a direct application of the multiplication rule. To form a permutation, you start by choosing an element for the first position from the total of n elements. Next, you choose an element for the second position from the remaining $n-1$ elements, then for the third position from the remaining $n-2$ elements, and continue this way until all positions are filled. Such arrangements are often called linear permutations.

Example 5.9 It is said that any shuffling of a deck of card has only happened once in history. This is because the number of ways to shuffle a deck of 52 cards is 52!, which is an astronomically large number.

$$52! \approx 8.07 \times 10^{67}$$

Example 5.10 Suppose you have 5 different books on a shelf. You want to rearrange them in a different order. The number of ways to rearrange the books is

$$5! = 5 \times 4 \times 3 \times 2 \times 1 = 120$$

There are cases where we are only interested in arranging a subset of elements from a larger set. The formula for counting these arrangements also derives from the multiplication rule.

Definition 5.6 (Permutations of Subsets)

The number of permutations of subsets of r elements selected from a set of n different elements is

$$P_r^n = n \times (n-1) \times (n-2) \times \cdots \times (n-r+1) = \frac{n!}{(n-r)!}$$

♣

Example 5.11 Suppose you have 5 different books on a shelf, and you want to rearrange 3 of them in a different order. The number of ways to rearrange the 3 books is

$$P_3^5 = 5 \times 4 \times 3 = 60$$

Example 5.12 There are 10 entries in a contest. Only three will win, 1^{st}, 2^{nd}, or 3^{rd} prize. What are the possible results?

Solution: The number of ways to award the prizes is the number of permutations of 3 objects selected from 10, which is

$$P_3^{10} = \frac{10!}{(10-3)!} = \frac{10!}{7!} = 10 \times 9 \times 8 = 720$$

Therefore, there are 720 possible outcomes for awarding the prizes. ◀

Combinations

When the order of selection does not matter, we use the concept of combinations. Combinations are used when we are interested in selecting a subset of elements from a larger set without regard to the order in which they are selected. Let is start out with a couple of examples to illustrate the concept of combinations.

Example 5.13 Suppose you have a group of 5 students: Alice, Bob, Charlie, David, and Eve. You need to select 2 of them to form a study group. The order in which you select the students does not matter. The possible combinations are:

- Alice and Bob
- Alice and Charlie
- Alice and David
- Alice and Eve
- Bob and Charlie
- Bob and David
- Bob and Eve
- Charlie and David
- Charlie and Eve
- David and Eve

The order of the students in the study group does not matter, so the combinations are considered identical.

Example 5.14 Maria has three tickets for a concert. She'd like to use one of the tickets herself. She could then offer the other two tickets to any of four friends (Ann, Beth, Chris, Dave). How many ways can 2 people be selected from 4 to go to a concert?

Example 5.15 A circuit board has four different locations in which a component can be placed. If three identical components are to be placed on the board, how many different designs are possible?

Solution: Since you can only place one component in each slot, placing a component in any slot immediately restricts the choices for the next component.

1. Fill slots 1, 2, and 3.
2. Fill slots 1, 2, and 4.
3. Fill slots 1, 3, and 4.
4. Fill slots 2, 3, and 4.

◀

These examples illustrate the concept of combinations, where the order of selection does not matter. The formula for combinations is derived from the permutation formula by dividing out the number of ways to arrange the r elements.

Definition 5.7 (Combinations)

The number of combinations of r elements selected from a set of n different elements is given by

$$C_r^n = \binom{n}{r} = \frac{n!}{r!(n-r)!}$$

♣

This is also sometimes referred to as the **binomial coefficient**, denoted by $\binom{n}{r}$, which is read as "n choose r". It is called the binomial coefficient because it appears in the binomial theorem, which expands the powers of a binomial expression:

Theorem 5.2

In algebra, the binomial coefficient is used to expand powers of binomials. According to the binomial theorem

$$(a+b)^n = \sum_{k=0}^{n} \binom{n}{k} a^k b^{n-k}$$

♡

The theorem states that the expansion of the binomial expression $(a+b)^n$ is the sum of the terms $\binom{n}{k}a^k b^{n-k}$ for $k = 0, 1, 2, \ldots, n$. The binomial coefficient $\binom{n}{k}$ gives the number of ways to choose k elements from a set of n elements. We will place no more emphasis on the binomial theorem here, but it is a fundamental concept in algebra and combinatorics, and is widely used in probability theory.

5.4 Probability Basics

In this section, we explore probability within discrete sample spaces — those with a finite or countably infinite set of outcomes.

Probability quantifies the likelihood or chance that an outcome of a random experiment will occur. For instance, when you hear, "The chance of rain today is 30%," it expresses our belief about the likelihood of rain. Probabilities are numbers assigned to outcomes, ranging from 0 to 1 (or equivalently, from 0% to 100%). A probability of 0 means the outcome will not happen, while a probability of 1 means it will happen for sure.

Probabilities can be interpreted in different ways:

- **Objective (or Classical) Probability**: Often referred to as classical probability, this approach is used when outcomes are equally likely, such as in rolling a fair die or flipping a coin. Probabilities are assigned based on the assumption that each outcome has an equal chance of occurring. For example, when rolling a fair six-sided die, the probability of rolling a 3 is $\frac{1}{6}$ because there are 6 equally likely outcomes (1, 2, 3, 4, 5, 6), and only one of them is a 3. The probability is the same for all observers.
- **Relative Frequency (Empirical Probability)**: Empirical probability is based on observations from experiments rather than theoretical calculations. For example, if a software tester runs a stress test on a server 100 times and it crashes 7 times, the empirical probability of a crash is $\frac{7}{100} = 0.07$. This approach relies on actual data rather than assumptions or intuition.
- **Subjective Probability**: This reflects our personal belief or degree of confidence in an outcome. Different people might assign different probabilities to the same event based on their knowledge or perspective. You and your friends discuss Denmark's chances of winning the World Cup. Based on recent performance and team strength, you estimate a 10% chance. However, a more optimistic friend assigns a 20% chance, while another gives only 5%, considering stronger competitors. This illustrates subjective probability, where each person's estimate varies based on personal beliefs and biases rather than objective data.

When assigning probabilities, it's essential that the sum of all probabilities in an experiment equals 1, ensuring consistency with the relative frequency interpretation.

We start by establishing the Axioms of Probability, which lay the foundation for how probabilities are assigned to events. These axioms define the basic properties that every probability measure must satisfy.

Axiom 5.1 (Axioms of Probability)

- **Axiom 1:** For any event A, $0 \leq P(A) \leq 1$.
- **Axiom 2:** Probability of the sample space S is $P(S)=1$.
- **Axiom 3:** If $A_1, A_2, A_3, \cdots$ are disjoint events, then $P\left(A_1 \cup A_2 \cup A_3 \cdots\right)=P\left(A_1\right)+P\left(A_2\right)+P\left(A_3\right)+\cdots$

♡

The property that $0 \leq P(A) \leq 1$ is equivalent to the requirement that a relative frequency must be between 0 and 1. The property that $P(S)=1$ is a consequence of the fact that an outcome from the sample space occurs on every trial of an experiment. Consequently, the relative frequency of S is 1. Property 3 implies that if the events A_1 and A_2 have no outcomes in common, the relative frequency of outcomes in $A_1 \cup A_2$ is the sum of the relative frequencies of the outcomes in A_1 and A_2.

In the next sections we will see more about the probability of events and how to calculate them.

Probability of an Event

The probability of an event is a measure of the likelihood that the event will occur. It is denoted by $P(A)$, where A is the event. The probability of an event ranges from 0 to 1, where 0 indicates that the event will not occur, and 1 indicates that the event will occur for sure.

Definition 5.8 (Probability of an Event)

The probability of an event A, denoted by $P(A)$, is the likelihood that event A will occur. It is defined as the ratio of the number of favorable outcomes to the total number of outcomes in the sample space.

$$P(A)=\frac{\text{Number of favorable outcomes}}{\text{Total number of outcomes}}$$

♣

Example 5.16 Suppose you are testing a software module with 10 different test cases. Out of these, 3 test cases are known to fail due to a bug. If you randomly select one test case to run, what is the probability that the selected test case will fail?

Solution: Here, the event A is "the test case fails."

- Number of favorable outcomes (failing test cases) $=3$
- Total number of outcomes (total test cases) $=10$

Using the formula:

$$P(A)=\frac{\text{Number of favorable outcomes}}{\text{Total number of outcomes}}=\frac{3}{10}=0.3$$

Therefore, the probability that a randomly selected test case will fail is 0.3 , indicating that there is a 30%

chance of failure.

◀

Example 5.17 Imagine a software development environment where you have 50 files, consisting of 20 Python scripts, 15 Java files, and 15 configuration files. If you randomly select one file to edit, what is the probability that the file is a Python script?

Solution: Here, the event A is "the selected file is a Python script."

- Number of favorable outcomes (Python scripts) $= 20$
- Total number of outcomes (total files) $= 50$

Using the formula:

$$P(A) = \frac{\text{Number of favorable outcomes}}{\text{Total number of outcomes}} = \frac{20}{50} = 0.4$$

Thus, the probability of selecting a Python script is 0.4 , meaning there is a 40% chance of choosing a Python file from the set.

◀

5.5 Probability of Joint Events and Set Operations

Joint events are formed by applying basic set operations to individual events. Commonly, we encounter unions of events, such as $A \cup B$; intersections of events, such as $A \cap B$; and complements of events, such as $\overline{A}$. These combined events are often of particular interest, and their probabilities can frequently be derived from the probabilities of the individual events that compose them. Understanding these set operations is essential for accurately calculating the probability of joint events. In this section, we will explore how unions of events and other set operations can be used to determine the probabilities of more complex events.

When dealing with events, the intersection represents `AND` while the union represents `OR` The probability of the intersection of events A and B, denoted as $P(A \cap B)$, can also be expressed as $P(A, B)$ or $P(AB)$:

Notation

- The probability of the intersection of events A and B is denoted as $P(A \cap B)$, $P(A, B)$, or $P(AB)$.
- The probability of the union of events A and B is denoted as $P(A \cup B)$.
- The probability of the complement of event A is denoted as $P(\overline{A})$, $P(A')$, or $P(A^c)$.

From the axioms of probability, we can derive the following rules of probabilities:

Theorem 5.3 (Rules of Probability)

- **Complement Rule:** The probability of the complement of event A is

  $P(\overline{A}) = 1 - P(A)$
- **Empty Set Rule:** The probability of the empty set is 0, i.e.,

  $P(\emptyset) = 0$
- **Addition Rule:** For any two events A and B, the probability of the union of events A and B is given by

  $P(A \cup B) = P(A) + P(B) - P(A \cap B)$
- **Difference Rule:** The probability of the difference between events A and B is given by

  $P(A - B) = P(A) - P(A \cap B)$
- **Subset Rule:** If A is a subset of B ($A \subset B$), then

  $P(A) \leq P(B)$

  .

♡

We can obtain the Complement Rule by noting:

$$\begin{aligned} 1 &= P(S) && \text{(axiom 2)} \\ &= P(A \cup \overline{A}) && \text{(definition of complement)} \\ &= P(A) + P(\overline{A}) && \text{(since } A \text{ and } \overline{A} \text{ are disjoint)} \end{aligned}$$

Since $\emptyset = \overline{S}$, we can apply part the Complement Rule to deduce that $P(\emptyset) = 1 - P(S) = 0$. This is intuitive because, by definition, an event occurs when the outcome of the random experiment is part of that event. However, since the empty set contains no elements, no outcome of the experiment can ever belong to it, making its probability zero.

The Difference Rule can be obtained by showing that $P(A) = P(A \cap B) + P(A - B)$. Note that the two sets $A \cap B$ and $A - B$ are disjoint and their union is A. Thus, by the third axiom of probability

$$\begin{aligned} P(A) &= P\left((A \cap B) \cup (A - B)\right) && \text{(since } A = (A \cap B) \cup (A - B)\text{)} \\ &= P(A \cap B) + P(A - B) && \text{(since } A \cap B \text{ and } A - B \text{ are disjoint)} \end{aligned}$$

The Addition Rule we obtain by noting that A and $B - A$ are disjoint sets and their union is $A \cup B$. Thus,

$$\begin{aligned} P(A \cup B) &= P(A \cup (B - A)) && \text{(since } A \cup B = A \cup (B - A)\text{)} \\ &= P(A) + P(B - A) && \text{(since } A \text{ and } B - A \text{ are disjoint)} \\ &= P(A) + P(B) - P(A \cap B) && \text{(by part the Difference Rule)} \end{aligned}$$

And finally the Subset Rule is a direct consequence of the fact that if $A \subset B$, then B can be written as the union of A and $B - A$. Since A and $B - A$ are disjoint, we have $P(B) = P(A) + P(B - A) \geq P(A)$.

We conclude this section with a few examples illustrating the application of these rules to calculate probabilities of joint events.

Example 5.18 A company has bid on two large construction projects. The company president believes that the probability of winning the first contract is 0.6, the probability of winning the second contract is 0.4, and the probability of winning both contracts is 0.2.

(a) What is the probability that the company wins at least one contract?
(b) What is the probability that the company wins the first contract but not the second contract?
(c) What is the probability that the company wins neither contract?
(d) What is the probability that the company wins exactly one contract?

Solution: Let A be the event that the company wins the first contract, and B be the event that the company wins the second contract. Given:

- $P(A) = 0.6$
- $P(B) = 0.4$
- $P(A \cap B) = 0.2$

(a) The probability that the company wins at least one contract is the probability of the union of events A and B. Using the Addition Rule:

$$\begin{aligned} P(A \cup B) &= P(A) + P(B) - P(A \cap B) \\ &= 0.6 + 0.4 - 0.2 \\ &= 0.8 \end{aligned}$$

Therefore, the probability that the company wins at least one contract is 0.8.

(b) The probability that the company wins the first contract but not the second contract is the probability of the difference between events A and B. Using the Difference Rule:

$$\begin{aligned} P(A - B) &= P(A) - P(A \cap B) \\ &= 0.6 - 0.2 \\ &= 0.4 \end{aligned}$$

Therefore, the probability that the company wins the first contract but not the second contract is 0.4.

(c) The probability that the company wins neither contract is the probability of the complement of the union of events A and B. Using the Complement Rule:

$$\begin{aligned} P(\overline{A \cup B}) &= 1 - P(A \cup B) \\ &= 1 - 0.8 \\ &= 0.2 \end{aligned}$$

Therefore, the probability that the company wins neither contract is 0.2.

(d) The probability that the company wins exactly one contract is the probability of the difference between the union of events A and B and the intersection of events A and B. Using the Difference Rule:

$$\begin{aligned} P((A \cup B) - (A \cap B)) &= P(A \cup B) - P(A \cap B) \\ &= 0.8 - 0.2 \\ &= 0.6 \end{aligned}$$

So, the probability that the company wins exactly one contract is 0.6.

◀

Example 5.19

- There is a 60 percent chance that it will rain today.
- There is a 50 percent chance that it will rain tomorrow.
- There is a 30 percent chance that it does not rain either day.

(a) The probability that it will rain today or tomorrow
(b) The probability that it will rain today and tomorrow.
(c) The probability that it will rain today but not tomorrow.
(d) The probability that it either will rain today or tomorrow, but not both.

Solution:

(a) Let A be the event that it rains today, and B be the event that it rains tomorrow.
Given:

$$\begin{aligned} P(A \cup B) &= 1 - P\left(\overline{(A \cup B)}\right) && \text{by the Complement Rule} \\ &= 1 - P\left(\overline{A} \cap \overline{B}\right) && \text{by De Morgan's Law} \\ &= 1 - 0.3 \\ &= 0.7 \end{aligned}$$

Therefore, the probability that it will rain today or tomorrow is 0.7.

(b) The probability that it will rain today and tomorrow: this is $P(A \cap B)$. To find this we note that

$$\begin{aligned} P(A \cap B) &= P(A) + P(B) - P(A \cup B) \\ &= 0.6 + 0.5 - 0.7 \\ &= 0.4 \end{aligned}$$

(c) The probability that it will rain today but not tomorrow: this is $P\left(A \cap \overline{B}\right)$.

$$\begin{aligned} P(A \cap \overline{B}) &= P(A - B) \\ &= P(A) - P(A \cap B) \\ &= 0.6 - 0.4 \\ &= 0.2 \end{aligned}$$

(d) The probability that it either will rain today or tomorrow but not both: this is $P(A-B)+P(B-A)$.
We have already found $P(A - B) = .2$. Similarly, we can find $P(B - A)$:

$$\begin{aligned} P(B - A) &= P(B) - P(B \cap A) \\ &= 0.5 - 0.4 \\ &= 0.1 \end{aligned}$$

Thus,

$$\begin{aligned} P(A - B) + P(B - A) &= 0.2 + 0.1 \\ &= 0.3 \end{aligned}$$

◀

Chapter 6 Conditional Probability and Bayes' Theorem

In this section, we introduce more fundamental concepts of probability. We begin with **conditional probability**, which assesses the likelihood of an event occurring *given* that another event has already taken place. Building on this, we introduce **the multiplication rule**, a key principle for determining the probability of multiple events happening in sequence. Next, we explore **the law of total probability**, which allows us to break down and calculate probabilities across different scenarios or partitions of the sample space. To further distinguish how events interact, we examine **dependent and independent events**, clarifying how the occurrence of one event influences or does not influence another.

Finally, we look into **Bayes' Theorem**, a powerful tool for updating probabilities in light of new evidence.

6.1 Conditional Probability

Consider the following scenario: in a particular population, 5% of individuals have a specific medical condition. Therefore, the probability that a randomly selected person has the condition is 5%:

$$P(D) = 0.05,$$

where D represents the event that a person has the disease. This probability $P(D)$ is known as the *prior probability*, as it reflects our initial belief about the likelihood of the event before any additional information is obtained.

Now, imagine selecting a random person and being informed that they have tested positive for the disease. With this additional information, how should we update the probability that the person actually has the disease? In other words, what is the probability that a person has the disease given that they tested positive? Let T denote the event that a person tests positive. This conditional probability is expressed as:

$$P(D \mid T),$$

which represents the probability of D occurring given that T has occurred. This updated probability $P(D \mid T)$ is known as the *posterior probability*, as it reflects our revised belief about the likelihood of the event after taking the new evidence into account. Intuitively, it is reasonable to expect that $P(D \mid T)$ is greater than the prior probability $P(D)$. However, what is the exact value of $P(D \mid T)$? Before introducing a general formula, let us consider a simple example.

Example 6.1 I roll a fair die. Let A be the event that the outcome is a prime number, i.e., $A = \{2, 3, 5\}$. Also, let B be the event that the outcome is greater than or equal to 3, i.e., $B = \{3, 4, 5\}$.

(a) What is the probability of A, $P(A)$?

(b) What is the probability of A given B, $P(A \mid B)$?

Solution:

(a) The probability of A is the number of outcomes in A divided by the total number of outcomes. Since there are 3 prime numbers and 6 possible outcomes, we have:

$$P(A) = \frac{3}{6} = \frac{1}{2}$$

(b) The probability of A given B is the number of outcomes in the intersection of A and B divided by the number of outcomes in B. Since the outcomes in the intersection of A and B are $\{3, 5\}$ and the outcomes in B are $\{3, 4, 5\}$, we have:

$$P(A \mid B) = \frac{|A \cap B|}{|B|} = \frac{|\{3, 5\}|}{|\{3, 4, 5\}|} = \frac{2}{3}$$

◀

Having understood the basic example, we can now generalize the approach to derive a more universal formula for conditional probability. Starting from the specific case, we can manipulate the expression by dividing both the numerator and the denominator by the total number of possible outcomes, $|S|$, as shown below:

$$P(A \mid B) = \frac{|A \cap B|}{|B|} = \frac{\frac{|A \cap B|}{|S|}}{\frac{|B|}{|S|}} = \frac{P(A \cap B)}{P(B)}$$

Explanation:

- **Numerator:** $|A \cap B|$ represents the number of outcomes where both events A and B occur. Dividing by $|S|$ converts this count into a probability, $P(A \cap B)$.
- **Denominator:** $|B|$ is the number of outcomes where event B occurs. Similarly, dividing by $|S|$ yields the probability of event B, denoted as $P(B)$.

Thus, the conditional probability $P(A \mid B)$ can be expressed as the ratio of the joint probability of A and B to the probability of B.

While the above derivation assumes a finite sample space with equally likely outcomes, the resulting formula is remarkably general. It holds true regardless of whether the sample space is finite or infinite, and whether the outcomes are equally likely or not. This universality makes the formula a cornerstone of probability theory.

Definition 6.1 (Conditional Probability)

For any two events A and B with $P(B) > 0$, the conditional probability of A given B is defined as:

$$P(A \mid B) = \frac{P(A \cap B)}{P(B)}, \qquad P(B) > 0$$

♣

Here is the intuition behind the formula. When we know that event B has occurred, we effectively eliminate all outcomes that are not part of B. This reduction transforms our original sample space into the subset B.

Within this new, restricted sample space B, the only way for event A to occur is if the outcome lies in the intersection of A and B, denoted by $A \cap B$. To determine the conditional probability $P(A \mid B)$, we calculate the ratio of the probability of both A and B occurring to the probability of B occurring alone. Mathematically, this is expressed as:

$$P(A \mid B) = \frac{P(A \cap B)}{P(B)}$$

This formulation ensures that the probabilities within the new sample space B are normalized, meaning the total probability sums to 1. For instance, consider the conditional probability of B given B:

$$P(B \mid B) = \frac{P(B \cap B)}{P(B)} = \frac{P(B)}{P(B)} = 1$$

This result intuitively confirms that if B has occurred, the probability of B occurring is certain, i.e., 1.

Note that the conditional probability $P(A \mid B)$ is undefined when $P(B) = 0$. This scenario implies that event B never occurs. Since conditional probability relies on the occurrence of B, if B has a probability of zero, there are no outcomes in the sample space to condition upon. Therefore, discussing the probability of A given B becomes meaningless in this context.

We can also formulate the axioms of probability in terms of conditional probability. The axioms of probability are as follows:

Axiom 6.1 (Axioms of Conditional Probability)

- **Axiom 1:** For any event A, $0 \leq P(A \mid B) \leq 1$.
- **Axiom 2:** Conditional probability of B given B is 1, i.e., $P(B \mid B) = 1$.
- **Axiom 3:** If $A_1, A_2, A_3, \cdots$ are disjoint events, then $P(A_1 \cup A_2 \cup A_3 \cdots \mid B) = P(A_1 \mid B) + P(A_2 \mid B) + P(A_3 \mid B) + \cdots$

♡

And we are also able to derive other rules of conditional probability from these axioms:

Theorem 6.1 (Rules of Conditional Probability)

- **Complement Rule:** The probability of the complement of event A given C is
  $$P\left(\overline{A} \mid C\right) = 1 - P(A \mid C)$$
- **Empty Set Rule:** The probability of the empty set given some event C is 0, i.e.,
  $$P(\emptyset \mid C) = 0$$
- **Addition Rule:** For any two events A and B, the probability of the union of events A and B given another event C is
  $$P(A \cup B \mid C) = P(A \mid C) + P(B \mid C) - P(A \cap B \mid C)$$
- **Difference Rule:** The probability of the difference between events A and B given another event C is
  $$P(A - B \mid C) = P(A \mid C) - P(A \cap B \mid C)$$
- **Subset Rule:** If $A \subset B$, then
  $$P(A \mid C) \leq P(B \mid C)$$

♡

The following example illustrates the application of conditional probability and introduces the concept of **contingency tables** as well as **false positives** and **false negatives**.

Example 6.2 A researcher aims to assess the effectiveness of a diagnostic test designed to detect renal disease in patients with high blood pressure. To achieve this, she conducts the test on a sample of 137 patients, categorized as follows:

- **67 patients** with a confirmed diagnosis of renal disease.
- **70 patients** who are known to be healthy (i.e., do not have renal disease).

The diagnostic test produces one of two possible outcomes for each patient:

- **Positive**: Indicates that the patient has renal disease.

- **Negative**: Indicates that the patient does not have renal disease.

The findings are summarised in the following contingency table:

	Test Results		
Truth	Positive	Negative	Total
Renal Disease	44	23	67
Healthy	10	60	70
Total	54	83	137

In this experiment:

- **True Positives**: Patients who have renal disease and tested positive.
- **False Negatives**: Patients who have renal disease but tested negative.
- **False Positives**: Healthy patients who tested positive.
- **True Negatives**: Healthy patients who tested negative.

Determine the following probabilities:

(a) The Probability of Renal Disease, $P(D)$.
(b) The Probability of Renal Disease, $P(T^+)$.
(c) The Probability of Renal Disease, $P(T^-)$.
(d) If a person has renal disease, what is the probability that they test positive for the disease?
(e) Determine the probability that a patient has renal disease given a positive test result, $P(\text{Renal Disease} \mid \text{Positive})$.
(f) Determine the probability that a patient does not have renal disease given a negative test result, $P(\text{Healthy} \mid \text{Negative})$.
(g) Assess the overall accuracy of the diagnostic test.

labelex:renal-disease

Solution: Using the contingency table, we can calculate the following probabilities:

(a) **Probability of Renal Disease,** $P(D)$:

$$P(D) = \frac{\text{Number of patients with renal disease}}{\text{Total number of patients}}$$
$$= \frac{67}{137} \approx 0.4883 \text{ or } 48.83\%$$

(b) **Probability of a Positive Test,** $P(T^+)$:

$$P(T^+) = \frac{\text{Number of positive tests}}{\text{Total number of patients}}$$
$$= \frac{54}{137} \approx 0.3942 \text{ or } 39.42\%$$

(c) **Probability of a Negative Test,** $P(T^-)$:

$$P(T^-) = \frac{\text{Number of negative tests}}{\text{Total number of patients}}$$
$$= \frac{80}{137} \approx 0.6058 \text{ or } 60.58\%$$

(d) **Probability of a Positive Test Given Renal Disease,** $P(T^+ \mid D)$:

$$P(T+ \mid D) = \frac{44}{67} \approx 0.6567$$

(e) **Probability of Renal Disease Given a Positive Test,** $P(D \mid T^+)$:

$$\begin{aligned} P(D \mid T^+) &= \frac{P(D \cap T^+)}{P(T^+)} \\ &= \frac{44}{54} = \frac{22}{27} \approx 0.8148 \end{aligned}$$

(f) **Probability of Being Healthy Given a Negative Test,** $P(\textbf{Healthy} \mid T^-)$:

$$\begin{aligned} P(\text{Healthy} \mid T^-) &= \frac{P(\text{Healthy} \cap T^-)}{P(T^-)} \\ &= \frac{60}{83} \approx 0.7229 \end{aligned}$$

(g) **Overall Accuracy of the Diagnostic Test**: The overall accuracy of the diagnostic test is the proportion of correct diagnoses, i.e., the sum of true positives and true negatives divided by the total number of patients:

$$\text{Overall Accuracy} = \frac{44 + 60}{137} = \frac{104}{137} \approx 0.7591$$

The results of the analysis indicate that the diagnostic test has a high probability of correctly identifying patients with renal disease (81.48%). However, the test is less effective at identifying healthy patients, with a probability of 72.29%. The overall accuracy of the test is 75.91%, reflecting the proportion of correct diagnoses across all patients. ◀

Here's a classic probability puzzle known as the Two-Child Problem.This scenario has appeared in various forms in the literature, each with subtle twists that lead to different results. Before looking at the calculations, take a moment to make your own predictions — you might be surprised by the outcomes!

Example 6.3 Consider a family with two children, and we are interested in the possible gender combinations of the children. The sample space for this situation is:

$$S = \{(G, G), (G, O), (B, O), (O, O)\}$$

where G represents a girl and O represents a child that is not a girl. For simplicity, we assume that all four outcomes are equally likely.

(a) What is the probability that both children are girls given that the first child is a girl?

(b) We ask the father: "Do you have at least one daughter?" He responds "Yes!" Given this extra information, what is the probability that both children are girls? In other words, what is the probability that both children are girls given that we know at least one of them is a girl?

Solution: Let A be the event that both children are girls, i.e., $A = \{(G, G)\}$. Let B be the event that the first child is a girl, i.e., $B = \{(G, G), (G, O)\}$. Finally, let C be the event that at least one of the children is a girl, i.e., $C = \{(G, G), (G, O), (O, G)\}$. Since the outcomes are equally likely, we can write

$$\begin{aligned} P(A) &= \frac{1}{4} \\ P(B) &= \frac{2}{4} = \frac{1}{2} \\ P(C) &= \frac{3}{4} \end{aligned}$$

(a) What is the probability that both children are girls given that the first child is a girl? This is $P(A \mid B)$, thus we can write

$$\begin{aligned} P(A \mid B) &= \frac{P(A \cap B)}{P(B)} \\ &= \frac{P(A)}{P(B)} \quad (\text{ since } A \subset B) \\ &= \frac{\frac{1}{4}}{\frac{1}{2}} = \frac{1}{2} \end{aligned}$$

(b) What is the probability that both children are girls given that we know at least one of them is a girl? This is $P(A \mid C)$, thus we can write

$$\begin{aligned} P(A \mid C) &= \frac{P(A \cap C)}{P(C)} \\ &= \frac{P(A)}{P(C)} \quad (\text{ since } A \subset C) \\ &= \frac{\frac{1}{4}}{\frac{3}{4}} = \frac{1}{3} \end{aligned}$$

◀

Remark: Many people might intuitively guess that both $P(A \mid B)$ and $P(A \mid C)$ would be 50%. However, while $P(A \mid B) = 50\%$, $P(A \mid C)$ is only 33%. This illustrates how probability can be counterintuitive. The key is to recognize that event B is a subset of event C. Specifically, B excludes the outcome (B, G), which is included in C. As a result, C has more outcomes not in A than B, leading to a smaller $P(A \mid C)$ compared to $P(A \mid B)$.

6.2 Multiplication and Total Probability Rules

The multiplication rule is a fundamental principle in probability theory that allows us to calculate the probability of multiple events occurring in sequence. This rule is particularly useful when the events are dependent, meaning that the occurrence of one event influences the probability of the subsequent event.

Theorem 6.2 (Multiplication Rule)

For any two events A and B, the probability of both events occurring is given by:

$$P(A \cap B) = P(A)P(B \mid A) = P(B)P(A \mid B)$$

♡

Example 6.4 Suppose a software development team is testing a new feature. There are two stages of testing: Unit Testing (A) and Integration Testing (B). The probability that a bug is detected in Unit Testing is $P(A) = 0.3$. If a bug is detected during Unit Testing, the probability that it will also be detected in Integration Testing is $P(B)$ A$) = 0.7$. What is the probability that a bug is detected in both stages of testing?

Solution: Using the Multiplication Rule:

$$P(A \cap B) = P(A) \cdot P(B \mid A) = 0.3 \times 0.7 = 0.21$$

Therefore, the probability that a bug is detected in both Unit Testing and Integration Testing is 0.21 . ◀

Example 6.5 A company has implemented a two-layer security system to protect against network breaches: **Firewall (A)** and **Intrusion Detection System (IDS) (B)**. The probability that a breach attempt is detected by the Firewall is $P(A) = 0.4$. If the breach passes through the Firewall, the probability that it is detected by the IDS is $P(B \mid \overline{A}) = 0.6$. The probability that the breach is detected by both the Firewall and the IDS is $P(B \mid A) = 0.8$.

(a) What is the probability that a breach is detected by at least one of the security layers?
(b) What is the probability that a breach is detected by both security layers?

Solution:

(a) **Probability of Detection by at Least One Layer:**
To find the probability that the breach is detected by at least one layer, we calculate the probability of a breach passing through both layers undetected and subtract it from 1.

$$P(\text{Undetected}) = P(\overline{A}) \cdot P(\overline{B} \mid \overline{A}) = (1 - 0.4) \cdot (1 - 0.6) = 0.6 \cdot 0.4 = 0.24$$

Therefore, the probability of detecting the breach with at least one layer is:

$$P(\text{Detected}) = 1 - P(\text{Undetected}) = 1 - 0.24 = 0.76$$

(b) **Probability of Detection by Both Layers:**
Using the Multiplication Rule:

$$P(A \cap B) = P(A) \cdot P(B \mid A) = 0.4 \cdot 0.8 = 0.32$$

The probability of detecting the breach with at least one layer is 0.76, while the probability of detecting the breach with by both the Firewall and the IDS is 0.32. This discrepancy highlights the importance of considering the dependencies between events when calculating probabilities.

◀

Remark: It might seem intuitive to some to calculate the joint probability of detection using $P(A \cap B) = P(A) \cdot P(B) = 0.4 \cdot 0.6 = 0.24$. However, this approach ignores the fact that the probability of detection by the IDS depends on the outcome of the Firewall. This is why we use the conditional probability $P(B \mid A)$, which properly accounts for the dependency between the two layers, yielding the correct result of 0.32 .

In some scenarios, the probability of an event depends on various conditions. By knowing the conditional probabilities under these different scenarios, we can determine the overall probability of the event. For instance, consider semiconductor manufacturing, where the probability that a chip causes a product failure when it is not contaminated is $P(\overline{B}) = 0.005$. On the other hand, if the chip is subjected to high levels of contamination, the probability of failure is $P(B \mid A) = 0.10$. Suppose that in a given production run, 20% of the chips are highly contaminated ($P(A) = 0.20$). What is the probability that a product using one of these chips fails?

The probability of failure depends on whether the chip was exposed to high contamination or not. For any event A, it can be decomposed into two mutually exclusive parts: one that intersects with B and another that intersects with the complement $\overline{B}$. Mathematically, we can express this as:

$$A = (B \cap A) \cup (\overline{B} \cap A)$$

This decomposition is visualized in the Venn diagram in Figure 6.1. Since B and $\overline{B}$ are mutually exclusive, their intersections with A are also mutually exclusive. Thus, using the rule for the probability of the union of mutually exclusive events and the multiplication rule, we can derive the total probability as follows:

$$P(A) = P(A \cap B) + P(A \cap \overline{B})$$

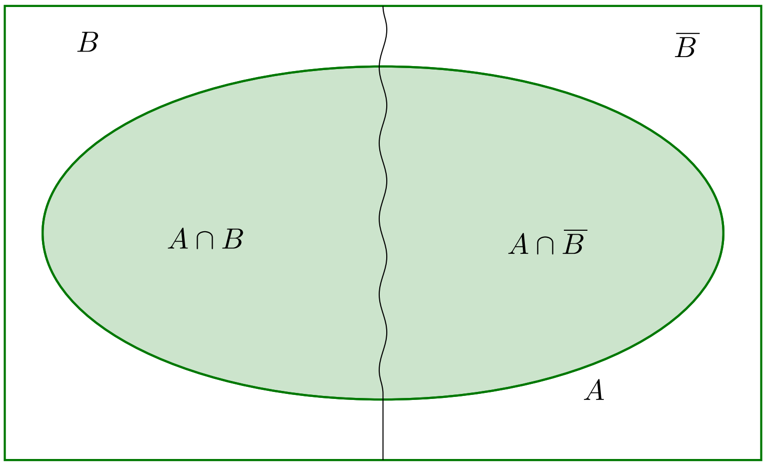


Figure 6.1: $P(A) = P(A \cap B) + P(A \cap \bar{B})$

We summarise this int he following theorem:

Theorem 6.3 (Law of Total Probability)

For any events A and B such that B and $\overline{B}$ form a partition of the sample space S, the total probability of event A is given by:

$$\begin{aligned} P(A) &= P(A \cap B) + P(A \cap \overline{B}) \\ &= P(B)P(A \mid B) + P(\overline{B})P(A \mid \overline{B}) \end{aligned}$$

For any event A and any partition of the sample space $B_1, B_2, \ldots, B_n$ such that $B_i \cap B_j = \emptyset$ for all $i \neq j$ and $\bigcup_{i=1}^{n} B_i = S$, the total probability of event A is given by:

$$P(A) = \sum_{i=1}^{n} P(A \mid B_i)P(B_i)$$

♡

Let us consider some examples.

Example 6.6 A company produces two types of products: **Product A** and **Product B**. The probability that a product is defective is $P(D \mid A) = 0.05$ for Product A and $P(D \mid B) = 0.10$ for Product B. The company manufactures 60% of its products as Product A and 40% as Product B. What is the probability that a randomly selected product is defective?

Solution: Using the Law of Total Probability:

$$P(D) = P(D \mid A)P(A) + P(D \mid B)P(B)$$

Substituting the given values:

$$P(D) = 0.05 \times 0.60 + 0.10 \times 0.40 = 0.03 + 0.04 = 0.07$$

Therefore, the probability that a randomly selected product is defective is 0.07. ◀

A graphical display of partitioning an event B among a collection of mutually exclusive and exhaustive events is shown in Figure 6.2. The event A is partitioned into five mutually exclusive events B_1, B_2, B_3, B_4, B_5, which together form the sample space S. The total probability of event A is calculated as the sum of the conditional probabilities of A given each partition B_i multiplied by the probability of each partition $P(B_i)$.

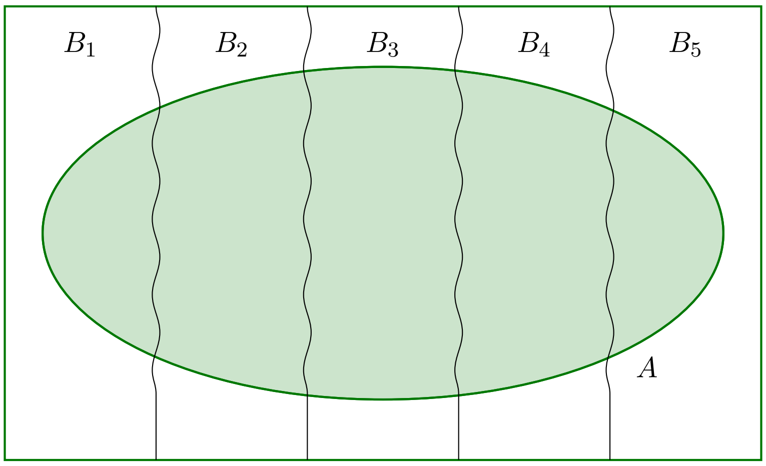


Figure 6.2: $P(A) = \sum_{i=1}^{5} P(A \mid B_i)P(B_i)$

Example 6.7 A university offers three types of courses: **Online**, **Hybrid**, and **In-Person**. The probability that a student fails a course is $P(F \mid \text{Online}) = 0.10$ for Online courses, $P(F \mid \text{Hybrid}) = 0.05$ for Hybrid courses, and $P(F \mid \text{In-Person}) = 0.02$ for In-Person courses. The university offers 50% of its courses as Online, 30% as Hybrid, and 20% as In-Person. What is the probability that a randomly selected student fails a course?

Solution: Using the Law of Total Probability:

$$P(F) = P(F \mid \text{Online})P(\text{Online}) + P(F \mid \text{Hybrid})P(\text{Hybrid}) + P(F \mid \text{In-Person})P(\text{In-Person})$$

Substituting the given values:

$$P(F) = 0.10 \times 0.50 + 0.05 \times 0.30 + 0.02 \times 0.20 = 0.05 + 0.015 + 0.004 = 0.069$$

Therefore, the probability that a randomly selected student fails a course is 0.069. ◀

6.3 Independence

Let A represent the event that it rains tomorrow, with $P(A) = \frac{1}{3}$. Additionally, suppose I toss a fair coin, and let B be the event that it lands heads up, so $P(B) = \frac{1}{2}$.

Now, consider the probability $P(A \mid B)$. What do you think it would be? You might intuitively guess that $P(A \mid B) = P(A) = \frac{1}{3}$, and you would be correct! The coin toss outcome has no influence on the weather forecast. This means that whether B occurs or not, the probability of A remains unchanged. This scenario illustrates the concept of independent events: two events are independent if the occurrence of one does not provide any information about the other.

Let's now formalize the definition of independence.

Definition 6.2 (Independence)

Two events are considered independent if the occurrence of one event does not affect the probability of the other event. In other words, the probability of one event does not depend on the occurrence of the other event. Two events A and B are independent if:

$$\begin{aligned} P(A \mid B) &= P(A) \\ P(B \mid A) &= P(B) \\ P(A \cap B) &= P(A)P(B) \end{aligned}$$

♣

In summary, independence can be understood in two equivalent ways: "Independence means that the probability of the intersection of two events can be found by simply multiplying their individual probabilities," or, alternatively, "Independence means that the conditional probability of one event given the other is the same as the original, unconditioned probability."

Lemma 6.1

If A and B are independent then

- A and $\overline{B}$ are independent,
- $\overline{A}$ and B are independent,
- $\overline{A}$ and $\overline{B}$ are independent.

♡

When dealing with the probability of the union of multiple independent events, $A_1, A_2, \ldots, A_n$, it is often easier to find the probability of their intersection than their union. In these situations, De Morgan's Law can be quite useful:

$$A_1 \cup A_2 \cup \cdots \cup A_n = \overline{\left(\overline{A_1} \cap \overline{A_2} \cap \cdots \cap \overline{A_n}\right)}$$

Using this relationship, we can express the probability of the union as:

$$\begin{aligned} P\left(A_1 \cup A_2 \cup \cdots \cup A_n\right) &= 1 - P\left(\overline{A_1} \cap \overline{A_2} \cap \cdots \cap \overline{A_n}\right) \\ &= 1 - P\left(\overline{A_1}\right) P\left(\overline{A_2}\right) \cdots P\left(\overline{A_n}\right) \quad \text{(since the } A_i\text{'s are independent)} \\ &= 1 - \left(1 - P\left(A_1\right)\right)\left(1 - P\left(A_2\right)\right) \cdots \left(1 - P\left(A_n\right)\right). \end{aligned}$$

Theorem 6.4 (Independence and DeMorgan's Law)

If $A_1, A_2, \cdots, A_n$ are independent then

$$P\left(A_1 \cup A_2 \cup \cdots \cup A_n\right) = 1 - \left(1 - P\left(A_1\right)\right)\left(1 - P\left(A_2\right)\right) \cdots \left(1 - P\left(A_n\right)\right)$$

♡

Warning! A common misconception is to confuse independence with disjointness. However, these are fundamentally different concepts. Two events, A and B, are disjoint if the occurrence of one prevents the occurrence of the other, i.e., $A \cap B = \emptyset$. In this case, knowing that A has occurred gives us complete information about B namely, that B cannot occur. This dependence means that disjoint events cannot be independent.

Concept	Description	Key Formulas
Disjoint	Events A and B cannot occur at the same time	$A \cap B = \emptyset$ $P(A \cup B) = P(A) + P(B)$
Independent	Occurrence of B gives no information about A	$P(A \mid B) = P(A)$ $P(B \mid A) = P(B)$ $P(A \cap B) = P(A) \cdot P(B)$

Table 6.1: Comparison of Disjoint and Independent Events.

We can extend the concept of independence to multiple events. A set of events $A_1, A_2, \ldots, A_n$ are considered independent if the occurrence of any subset of these events does not provide any information about the occurrence of the other events. Mathematically, this is expressed as:

Definition 6.3 (Independence of Multiple Events)

A set of events $A_1, A_2, \ldots, A_k$ are considered independent if the joint probability is equal to the product of the individual probabilities:

$$P(A_1 \cap A_2 \cap \cdots \cap A_k) = P(A_1) \cdot P(A_2) \cdots P(A_k)$$

♣

Example 6.8 Gambler's Fallacy

The **Gambler's Fallacy** is a common cognitive bias that arises when individuals believe that the outcome of a random event is influenced by previous outcomes. This fallacy is often observed in gambling scenarios, where individuals incorrectly assume that the probability of an event occurring is affected by past events.

A man tosses a fair coin eight times and observes whether the toss yields a head (H) or a tail (T) on each toss. Which of the following sequences of coin tosses is the man more likely to get a head (H) on his next toss? This one:

TTTTTTTT

or this one:

HHTHTTHH

The answer is neither as illustrated here:

$$\begin{aligned} P\left(H_9 \mid T_1 T_2 \ldots T_8\right) = \frac{P\left(H_9 \cap T_1 \cap T_2 \cap \cdots \cap T_8\right)}{P\left(T_1 \cap T_2 \cap \cdots \cap T_8\right)} &= \frac{\left(\frac{1}{2}\right)^9}{\left(\frac{1}{2}\right)^8} \\ &= \frac{1}{2} \end{aligned}$$

Try to avoid falling into the trap of the Gambler's Fallacy. For example, if a fair coin lands on tails eight times in a row, it's easy to think that a head is "due" on the next toss. However, each toss is independent, and the probability remains the same — there's still a 50% chance of heads or tails, regardless of past results.

6.4 Bayes' Theorem

We are now ready to introduce one of the most powerful tools in conditional probability: Bayes' rule (or theorem). This rule is particularly useful when we know $P(A \mid B)$ but want to find $P(B \mid A)$. Starting with the definition of conditional probability, we have:

$$P(A \mid B) \cdot P(B) = P(A \cap B) = P(B \mid A) \cdot P(A)$$

By dividing both sides by $P(A)$, we arrive at:

$$P(B \mid A) = \frac{P(A \mid B) \cdot P(B)}{P(A)}$$

This formula is famously known as Bayes' rule. In many cases, to calculate $P(A)$ in Bayes' rule, we need the law of total probability. Therefore, Bayes' rule is often presented in the form:

$$P(B \mid A) = \frac{P(A \mid B) \cdot P(B)}{P(A \mid B) \cdot P(B) + P(A \mid \overline{B}) \cdot P(\overline{B})}$$

or more generally

$$P\left(B_j \mid A\right) = \frac{P\left(A \mid B_j\right) \cdot P\left(B_j\right)}{\sum_i P\left(A \mid B_i\right) \cdot P\left(B_i\right)}$$

where $B_1, B_2, \ldots, B_n$ form a partition of the sample space.

Theorem 6.5 (Bayes' Theorem)

For any two events A and B, where $P(A) \neq 0$, we have

$$P(B \mid A) = \frac{P(A \mid B) \cdot P(B)}{P(A)} = \frac{P(A \mid B) \cdot P(B)}{P(A \mid B) \cdot P(B) + P(A \mid \overline{B}) \cdot P(\overline{B})}$$

If $B_1, B_2, B_3, \cdots$ form a partition of the sample space S, and A is any event with $P(A) \neq 0$, we have

$$P\left(B_j \mid A\right) = \frac{P\left(A \mid B_j\right) \cdot P\left(B_j\right)}{\sum_i P\left(A \mid B_i\right) \cdot P\left(B_i\right)}$$

♡

We start with a´n infamous example to highlight the importance of Bayes' Theorem.

Example 6.9 False Positive Paradox

Imagine a rare disease that affects about 1 in every 10,000 people. There is a test available to detect this disease, and while the test is highly accurate, it is not perfect. Specifically:

- The probability that the test shows a positive result (indicating the disease) when the person does not have the disease is 2%.
- The probability that the test shows a negative result (indicating no disease) when the person does have the disease is 1%.

Now, suppose a randomly selected person takes the test, and the result comes back positive. What is the probability that this person actually has the disease?

Solution: Let D be the event that the person has the disease, and let T^+ be the event that the test result is positive. We know:

$$P(D) = \frac{1}{10,000}$$
$$P\left(T^{+} \mid \overline{D}\right) = 0.02$$
$$P\left(\overline{T^{+}} \mid D\right) = 0.01$$

We want to compute $P(D \mid T^{+})$. Using Bayes' rule:

$$\begin{aligned} P(D \mid T^{+}) &= \frac{P(T^{+} \mid D)P(D)}{P(T^{+} \mid D)P(D) + P\left(T^{+} \mid \overline{D}\right) P\left(\overline{D}\right)} \\ &= \frac{(1-0.01) \times 0.0001}{(1-0.01) \times 0.0001 + 0.02 \times (1-0.0001)} \\ &= 0.0049 \end{aligned}$$

This result means there is less than half a percent chance that the person actually has the disease. Despite the positive test result, the low prevalence of the disease and the test's false positive rate contribute to this counterintuitive outcome. ◀

Example 6.10 Bayesian networks are commonly used on the websites of high-technology manufacturers to help customers quickly diagnose issues with their products. For instance, a printer manufacturer uses data from test results to identify potential causes of printer failures. Printer failures are primarily associated with three types of problems: hardware, software, and other issues (like connectors). The probabilities of these problems are as follows:

- Probability of a hardware issue: $P(H) = 0.1$
- Probability of a software issue: $P(S) = 0.6$
- Probability of another type of issue: $P(O) = 0.3$

The likelihood of a printer failing, given each type of problem, is:

- Probability of failure given a hardware issue: $P(F \mid H) = 0.9$
- Probability of failure given a software issue: $P(F \mid S) = 0.2$
- Probability of failure given another issue: $P(F \mid O) = 0.5$

Given that a customer experiences a printer failure and uses the manufacturer's website to diagnose the issue, what is the most likely cause of the problem?

Solution: To determine the most likely cause of the printer failure, we need to calculate the probability of each type of problem given that a failure has occurred. This involves using Bayes' theorem to compute the posterior probabilities.

Let F denote the event of a printer failure. We want to find:

$$P(H \mid F), \quad P(S \mid F), \quad P(O \mid F)$$

Step 1: Calculate the Total Probability of Failure

First, we use the law of total probability to find $P(F)$:

$$P(F) = P(F \mid H) \cdot P(H) + P(F \mid S) \cdot P(S) + P(F \mid O) \cdot P(O)$$

Substituting the given values:

$$P(F) = (0.9 \times 0.1) + (0.2 \times 0.6) + (0.5 \times 0.3)$$

$$P(F) = 0.09 + 0.12 + 0.15 = 0.36$$

Step 2: Apply Bayes' Theorem to Find the Posterior Probabilities

Now, apply Bayes' theorem for each problem type:

1. Probability of Hardware Problem Given Failure:
$$P(H \mid F) = \frac{P(F \mid H) \cdot P(H)}{P(F)} = \frac{0.9 \times 0.1}{0.36} = \frac{0.09}{0.36} = 0.25$$
2. Probability of Software Problem Given Failure:
$$P(S \mid F) = \frac{P(F \mid S) \cdot P(S)}{P(F)} = \frac{0.2 \times 0.6}{0.36} = \frac{0.12}{0.36} = 0.3333$$
3. Probability of Other Problems Given Failure:
$$P(O \mid F) = \frac{P(F \mid O) \cdot P(O)}{P(F)} = \frac{0.5 \times 0.3}{0.36} = \frac{0.15}{0.36} = 0.4167$$

Step 3: Interpret the Results

- Hardware Problem: 25% chance
- Software Problem: Approximately 33.33% chance
- Other Problem: Approximately 41.67% chance

Conclusion: Given a printer failure, the most likely cause is an Other Problem, such as connectors, with a probability of approximately 41.67%.

◀

Chapter 7 Linear Equations in Linear Algebra

In 1949, Harvard Professor Wassily Leontief used one of the earliest computers, the Mark II, to solve a system of linear equations that modelled the U.S. economy. He divided the economy into 500 sectors, like coal, automotive, and communications, and described how each sector interacted using linear equations. Due to hardware limitations, he reduced the problem to 42 equations, which took the Mark II 56 hours to solve. This effort marked a milestone in computational applications, earning Leontief the 1973 Nobel Prize.

Leontief's work showcased how linear algebra, combined with computing, could solve large-scale problems — an idea that resonates even more today in the realm of software engineering. Linear algebra is fundamental to many modern technologies, particularly in Machine Learning (ML) and Artificial Intelligence (AI). From powering recommendation systems to optimising neural networks, linear algebra is at the core of algorithms that drive today's intelligent systems.

In ML and AI, large datasets are transformed into matrices, where linear algebra techniques like matrix factorisation, eigenvalue decomposition, and singular value decomposition (SVD) are used to extract insights and make predictions. Neural networks, the backbone of AI, rely on linear algebra to propagate data through layers and adjust weights during training. As a software engineer, mastering these concepts in linear algebra equips you to build scalable, intelligent systems capable of tackling today's most complex computational problems.

With the exponential growth in data and computing, the significance of linear algebra in software development, especially in AI and ML, continues to rise. It is a vital tool that bridges theoretical mathematics with real-world applications in tech.

7.1 Systems of Linear Equations

Linear equations and their systems form the foundation of linear algebra, a critical area in mathematics with extensive applications in software engineering. From computer graphics to machine learning algorithms, understanding how to model and solve linear systems is essential for developing efficient and effective software solutions. We begin with a definition:

Definition 7.1 (Linear Equations)

A linear equation in the variables $x_1, x_2, \ldots, x_n$ is an equation that can be written in the form

$$a_1x_1 + a_2x_2 + \cdots + a_nx_n = b$$

where b and the coefficients $a_1, \ldots, a_n$ are constants. ♣

Linear equations represent straight lines, planes, and hyperplanes in various dimensions, making them useful for modeling relationships in data and algorithms.

Example 7.1 A line in two-dimensional space given by $y = mx + b$ is a linear equation. It can be rewritten to fit our standard form:

$$-mx + y = b$$

Here, $a_1 = -m$, $a_2 = 1$, and b is the constant term.

Example 7.2 The general equation of a plane in three-dimensional space is

$$ax + by + cz = d$$

where a, b, c, and d are constants. This equation is linear in the variables x, y, and z.

Definition 7.2 (Systems of Linear Equations)

A **system of linear equations** (also called a **linear system**) is a collection of one or more linear equations involving the same set of variables. ♣

Example 7.3 Consider the following system of linear equations in the variables x_1, x_2, x_3:

$$\begin{aligned} 2x_1 + 3x_2 + x_3 &= 3 \\ 7x_2 - 4x_3 &= 10 \\ x_3 &= 1 \end{aligned}$$

This system contains three equations with three unknowns.

To solve the system, we first note from the third equation that $x_3 = 1$. Substituting this into the second equation, we solve for x_2:

$$\begin{aligned} 7x_2 - 4(1) &= 10 \\ 7x_2 &= 14 \\ x_2 &= 2 \end{aligned}$$

Now, substituting $x_2 = 2$ and $x_3 = 1$ into the first equation, we solve for x_1:

$$\begin{aligned} 2x_1 + 3(2) + 1 &= 3 \\ 2x_1 &= -4 \\ x_1 &= -2 \end{aligned}$$

Thus, the solution to the system is $(x_1, x_2, x_3) = (-2, 2, 1)$.

Definition 7.3 (Solutions to a System of Linear Equations)

A **solution** of a linear system in the variables $x_1, x_2, \ldots, x_n$ is a list of numbers $(s_1, s_2, \ldots, s_n)$ that satisfies all equations of the system when substituted for the variables $x_1, x_2, \ldots, x_n$, respectively. The set of all possible solutions is called its **solution set**. Two linear systems are called **equivalent** if they have the same solution set. ♣

A system of linear equations has one of the following outcomes:

(i) **No solutions**, when the equations are inconsistent.
(ii) **Exactly one solution**, when there is a unique set of values satisfying all equations.
(iii) **Infinitely many solutions**, when there are multiple sets of values that satisfy the equations.

We say a system is **consistent** if it has either one or infinitely many solutions. We say a system is **inconsistent** if it has no solution. We formalise this later in this chapter.

Remark: Determining whether a system is consistent addresses the *existence* of solutions. If solutions exist, we may further explore the *uniqueness* of these solutions.

Representing Systems with Matrices

Matrices provide a compact and efficient way to represent and manipulate systems of linear equations, which is particularly beneficial in software applications involving large datasets.

Definition 7.4
A **matrix** is a rectangular array of numbers arranged in rows and columns. The **coefficient matrix** of a linear system contains only the coefficients of the variables, while the **augmented matrix** includes an additional column for the constants from the right-hand side of the equations. ♣

Example 7.4 For the linear system:

$$\begin{aligned} 2x_1 + 3x_2 + x_3 &= 3 \\ 7x_2 - 4x_3 &= 10 \\ x_3 &= 1 \end{aligned}$$

the coefficient matrix is:

$$\begin{bmatrix} 2 & 3 & 1 \\ 0 & 7 & -4 \\ 0 & 0 & 1 \end{bmatrix}$$

and the augmented matrix is:

$$\left[\begin{array}{ccc|c} 2 & 3 & 1 & 3 \\ 0 & 7 & -4 & 10 \\ 0 & 0 & 1 & 1 \end{array}\right]$$

The vertical line between the coefficient part and the augmented part is optional and is used to visually separate the coefficients from the constants.

Definition 7.5
The **size** of a matrix is defined by the number of its rows and columns, expressed as *rows* $\times$ *columns*. For instance, the coefficient matrix above is of size 3×3, and the augmented matrix is of size 3×4. ♣

Solving Systems using Augmented Matrices

We now illustrate how to solve a system of linear equations using augmented matrix notation. This process will be formalised in the next section but here we offer an example. This method streamlines the process of solving systems by focusing on matrix manipulations.

Example 7.5 Let r_1 denote the first row, r_2 the second row, and r_3 the third row.

$$\begin{bmatrix} 2 & 3 & 1 & 3 \\ 0 & 7 & -4 & 10 \\ 0 & 0 & 1 & 1 \end{bmatrix}$$

To simplify the system, we perform *elementary row operations*. Specifically, we adjust r_1 and r_2 as follows:

$$r_1 \mapsto r_1 - r_3,$$
$$r_2 \mapsto r_2 + 4r_3,$$

which gives:

$$\begin{bmatrix} 2 & 3 & 0 & 2 \\ 0 & 7 & 0 & 14 \\ 0 & 0 & 1 & 1 \end{bmatrix}$$

Next, we scale r_2 by $\frac{1}{7}$:

$$r_2 \mapsto \frac{1}{7} r_2, \quad \begin{bmatrix} 2 & 3 & 0 & 2 \\ 0 & 1 & 0 & 2 \\ 0 & 0 & 1 & 1 \end{bmatrix}$$

Now, we eliminate the 3 in the first row by performing:

$$r_1 \mapsto r_1 - 3r_2, \quad \begin{bmatrix} 2 & 0 & 0 & -4 \\ 0 & 1 & 0 & 2 \\ 0 & 0 & 1 & 1 \end{bmatrix}$$

Finally, we scale r_1 by $\frac{1}{2}$ to get:

$$r_1 \mapsto \frac{1}{2} r_1, \quad \begin{bmatrix} 1 & 0 & 0 & -2 \\ 0 & 1 & 0 & 2 \\ 0 & 0 & 1 & 1 \end{bmatrix}$$

We can now reinterpret the matrix as a linear system. From the augmented matrix, we find:

$$x_1 = -2,$$
$$x_2 = 2,$$
$$x_3 = 1.$$

This can be expressed compactly as:

$$(x_1, x_2, x_3) = (-2, 2, 1).$$

Thus, the solution is identical to the one we obtained through substitution.

Example 7.6 Determine if the following system is consistent:

$$\begin{aligned} x_2 + 4x_3 &= 2 \\ x_1 - 3x_2 + 2x_3 &= 6 \\ x_1 - 2x_2 + 6x_3 &= 9 \end{aligned}$$

Solution: First, we write the augmented matrix of the system:

$$\begin{bmatrix} 0 & 1 & 4 & 2 \\ 1 & -3 & 2 & 6 \\ 1 & -2 & 6 & 9 \end{bmatrix}$$

To simplify the matrix, we interchange r_1 and r_2:

$$r_1 \leftrightarrow r_2, \quad \begin{bmatrix} 1 & -3 & 2 & 6 \\ 0 & 1 & 4 & 2 \\ 1 & -2 & 6 & 9 \end{bmatrix}$$

Next, we eliminate the 1 in the first column of r_3 by performing:

$$r_3 \mapsto r_3 - r_1, \quad \begin{bmatrix} 1 & -3 & 2 & 6 \\ 0 & 1 & 4 & 2 \\ 0 & 1 & 4 & 3 \end{bmatrix}$$

Then, we eliminate the 1 in the second column of r_3:

$$r_3 \mapsto r_3 - r_2, \quad \begin{bmatrix} 1 & -3 & 2 & 6 \\ 0 & 1 & 4 & 2 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

The third row now corresponds to the equation:

$$0x_1 + 0x_2 + 0x_3 = 1$$

which simplifies to:

$$0 = 1$$

This is a contradiction, so the system is **inconsistent** and has no solution. ◀

Example 7.7 Give a solution of the following system (if one exists). Is it unique?

$$\begin{aligned} x_1 + 2x_2 + 3x_3 &= 4 \\ 3x_1 + 6x_2 + 9x_3 &= 12 \end{aligned}$$

Solution:

$$\begin{bmatrix} 1 & 2 & 3 & 4 \\ 3 & 6 & 9 & 12 \end{bmatrix}$$

We observe that r_2 is a multiple of r_1:

$$r_2 = 3r_1$$

To simplify the system, we perform the following elementary row operation to eliminate redundancy:

$$r_2 \mapsto r_2 - 3r_1,$$

which yields:

$$\begin{bmatrix} 1 & 2 & 3 & 4 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

The second row now consists entirely of zeros, indicating that it does not provide any new information. This means the system is underdetermined and has infinitely many solutions.

We can express the first equation in terms of x_1:

$$x_1 = 4 - 2x_2 - 3x_3$$

Letting x_2 and x_3 be free variables (parameters), we set:

$$x_2 = s, \quad x_3 = t, \quad \text{where } s, t \in \mathbb{R}$$

Substituting back, we find:

$$\begin{aligned} x_1 &= 4 - 2s - 3t, \\ x_2 &= s, \\ x_3 &= t. \end{aligned}$$

This can be expressed compactly as:

$$(x_1, x_2, x_3) = (4 - 2s - 3t,\ s,\ t), \quad \text{for all } s, t \in \mathbb{R}.$$

The system has infinitely many solutions parameterized by s and t. Therefore, the solution is not unique.◀

Example 7.8 Choose h and k such that the following system

$$\begin{aligned} x_1 - 3x_2 &= 1 \\ 2x_1 + hx_2 &= k \end{aligned}$$

has

(i) a unique solution,
(ii) many solutions, and
(iii) no solution.

Solution:

$$\begin{bmatrix} 1 & -3 & 1 \\ 2 & h & k \end{bmatrix}$$

To simplify the system, we perform an elementary row operation to eliminate x_1 from the second equation. Specifically, we adjust r_2 as follows:

$$r_2 \mapsto r_2 - 2r_1,$$

which yields:

$$\begin{bmatrix} 1 & -3 & 1 \\ 0 & h+6 & k-2 \end{bmatrix}$$

Now, we analyze the resulting system based on the value of $h + 6$.

Case 1: $h + 6 \neq 0$

When $h + 6 \neq 0$, we can solve for x_2 from the second equation:

$$(h + 6)x_2 = k - 2 \implies x_2 = \frac{k-2}{h+6}$$

Substituting x_2 back into the first equation:

$$x_1 - 3x_2 = 1 \implies x_1 = 1 + 3x_2 = 1 + 3\left(\frac{k-2}{h+6}\right)$$

Thus, we obtain a unique solution for x_1 and x_2. **The system has a unique solution when $h + 6 \neq 0$.**

Case 2: $h + 6 = 0$

When $h + 6 = 0$, i.e., $h = -6$, the second equation becomes:

$$0x_2 = k - 2$$

This simplifies to:

$$0 = k - 2$$

We have two subcases:

Subcase 2a: $k - 2 = 0$

If $k = 2$, the equation becomes $0 = 0$, which is always true. The second equation provides no new information, so the system reduces to:

$$x_1 - 3x_2 = 1$$

Here, x_2 is a free variable. Solving for x_1:

$$x_1 = 1 + 3x_2$$

The system has infinitely many solutions when $h = -6$ and $k = 2$.

Subcase 2b: $k - 2 \neq 0$

If $k \neq 2$, the equation becomes $0 = k - 2$, which is a contradiction since $0 \neq k - 2$. **Thus, the system has no solution when $h = -6$ and $k \neq 2$.**

Conclusion:

(i) **No solution** when $h = -6$ and $k \neq 2$.
(ii) **A unique solution** when $h \neq -6$.
(iii) **Infinitely many solutions** when $h = -6$ and $k = 2$.

◀

7.2 Elementary Row Operations and Echelon Forms

In examples 7.5-7.8, we performed a series of operations known as **elementary row operations**. These are fundamental transformations that simplify systems without changing their solution sets.

The three types of elementary row operations are:

- **Replacement:** Replace one row by the sum of itself and a multiple of another row.
- **Interchange:** Swap two rows.
- **Scaling:** Multiply all entries in a row by a nonzero constant.

Two matrices are called **row equivalent** if one can be transformed into the other through a sequence of elementary row operations.

Remark: Two linear systems are equivalent (i.e., they have the same solution set) if their augmented matrices are row equivalent. Performing elementary row operations on an augmented matrix does not change the solution set of the system.

As you may have noticed in the examples, the matrices resulting from these operations often have a special pattern — lots of zeros below certain entries. This isn't just a coincidence but a goal in the process. We call this the **echelon form** of a matrix. By organizing a matrix into echelon form, we:

- Make solving systems of linear equations easier.
- Systematically simplify the matrix, reducing the complexity of calculations.
- Reveal key properties about the system, like existence and number of solutions or whether certain equations are dependent.

Definition 7.6 (Echelon Forms)

A matrix is in **echelon form** if it satisfies the following conditions:

1. All zero rows are at the bottom.
2. Each leading entry of a row is in a column to the right of the leading entry of the row above it.
3. All entries in a column below a leading entry are zeros.

In addition to echelon form, a matrix may also be in **reduced row echelon form** (RREF), which has the following properties:

4. The leading entry in each nonzero row is 1.
5. Each leading 1 is the only nonzero entry in its column.

♣

We say that an echelon matrix U is an echelon form of the matrix A if U is row equivalent to A. Similarly, we say that a reduced echelon matrix U is the reduced echelon form of the matrix A if U is row equivalent to A.

The significance of putting the augmented matrix of a linear system in echelon form is explained by the following theorem.

Theorem 7.1 (Existence Theorem)

A linear system is consistent if and only if an echelon form of the augmented matrix has no row of the form

$$\begin{bmatrix} 0 & \ldots & 0 & b \end{bmatrix}$$

where b is nonzero.

♡

We saw an example of an inconsistent system in Example 7.6 because the echelon form of the augmented matrix has a row of the form $\left[\begin{array}{ccc|c} 0 & 0 & 0 & 1 \end{array}\right]$. This row indicates that the system has no solution.

Example 7.9 The augmented matrix of the linear system used as the main example in the preceding section,

$$\begin{bmatrix} 2 & 3 & 1 & 3 \\ 0 & 7 & -4 & 10 \\ 0 & 0 & 1 & 1 \end{bmatrix},$$

is already in echelon form. Since it has no row of the form mentioned in the theorem, we know immediately that this system is consistent. The leading entries are 2, 7, and 1, and all entries below them are zeros. This matrix is not in reduced row echelon form however because the leading entries are not all 1.

Recall that we performed a sequence of row operations on the preceding matrix to get

$$\begin{bmatrix} 1 & 0 & 0 & -2 \\ 0 & 1 & 0 & 2 \\ 0 & 0 & 1 & 1 \end{bmatrix}$$

which is in reduced echelon form. This allowed us easily to see the solutions of this system, which is the main advantage of putting the matrix in this form. This brings us to the following important theorem.

Theorem 7.2 (Uniqueness of Reduced Echelon Form)

Each matrix is row equivalent to one and only one reduced echelon matrix ♡

A pivot position in a matrix A is a location in A that corresponds to a leading 1 in the reduced echelon form of A. A pivot column is a column of A that contains a pivot position. Notice that you can obtain a pivot by scaling the leading entry of a row to be 1. Therefore, the pivot column is the column of the leading entry.

We are now ready to present the algorithm for solving a system of linear equations using matrices.

The Row Reduction Algorithm

Here we describe an algorithm for turning any matrix into an equivalent (reduced) echelon matrix. This algorithm is the foundation of solving systems of linear equations using matrices.

1. Begin with the leftmost nonzero column. This is a pivot column, with the pivot position at the top.
2. Select a nonzero entry in the pivot column as a pivot. If necessary, interchange rows to move this entry into the pivot position.
3. Use row replacement operations to create zeros in all positions below the pivot.
4. Apply steps 1-3 to the submatrix of all entries below and to the right of the pivot position. Repeat this process until there are no more nonzero rows to modify. (At this point we have reached an echelon form of the matrix.)
5. Beginning with the rightmost pivot and working upward and to the left, create zeros above each pivot using row operations. If a pivot is not 1, make it 1 by a scaling operation. (This step produces the reduced echelon form of the matrix.)

Solutions of Linear Systems

Let A be the coefficient matrix of a linear system. The pivot columns in the matrix correspond to what we call **basic variables**. The nonpivot columns correspond to what we call **free variables**.

Example 7.10 Suppose the augmented matrix of a linear system has been reduced to the following form:

$$\begin{bmatrix} \blacksquare & * & * & * & * \\ 0 & \blacksquare & * & * & * \\ 0 & 0 & 0 & \blacksquare & * \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

where $\blacksquare$ represents any nonzero number, and $*$ represents any number (including 0). The basic variables of this system are x_1, x_2, and x_4. The only free variable is x_3.

Theorem 7.3 (Uniqueness Theorem)

If a linear system is consistent, then the solution set contains either

(i) a unique solution, when there are no free variables, or

(ii) infinitely many solutions, when there is at least one free variable.

♡

Example 7.11 Suppose the following matrix is the augmented matrix of a linear system in the variables x_1, x_2, and x_3. Row reduce the matrix to echelon form to determine if it is consistent.

$$\begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \end{bmatrix}$$

If it is consistent, find the reduced echelon form and write the solution set using free variables as parameters.

Solution: To simplify the system, we perform *elementary row operations*.

Step 1: Eliminate the 5 in r_2

We adjust r_2 as follows:

$$r_2 \mapsto r_2 - 5r_1,$$

which gives:

$$\begin{bmatrix} 1 & 2 & 3 & 4 \\ 0 & -4 & -8 & -12 \\ 9 & 10 & 11 & 12 \end{bmatrix}$$

Step 2: Eliminate the 9 in r_3

We adjust r_3 as follows:

$$r_3 \mapsto r_3 - 9r_1,$$

which gives:

$$\left[\begin{array}{ccc|c} 1 & 2 & 3 & 4 \\ 0 & -4 & -8 & -12 \\ 0 & -8 & -16 & -24 \end{array}\right]$$

Step 3: Eliminate the -8 in r_3

We adjust r_3 again to eliminate the entry in the second column:

$$r_3 \mapsto r_3 - 2r_2,$$

which gives:

$$\begin{bmatrix} 1 & 2 & 3 & 4 \\ 0 & -4 & -8 & -12 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

Step 4: Scale r_2 to obtain a leading 1

We scale r_2 by $-\frac{1}{4}$:

$$r_2 \mapsto -\frac{1}{4}r_2,$$

which gives:

$$\left[\begin{array}{ccc|c} 1 & 2 & 3 & 4 \\ 0 & 1 & 2 & 3 \\ 0 & 0 & 0 & 0 \end{array}\right]$$

Step 5: Eliminate the 2 in r_1

We adjust r_1 as follows:

$$r_1 \mapsto r_1 - 2r_2,$$

which gives:

$$\left[\begin{array}{ccc|c} 1 & 0 & -1 & -2 \\ 0 & 1 & 2 & 3 \\ 0 & 0 & 0 & 0 \end{array}\right]$$

At this point, the matrix is in *reduced row-echelon form*.

Step 6: Interpret the matrix as a linear system

From the augmented matrix, we have:

$$\begin{aligned} r_1 &: \quad x_1 - x_3 = -2, \\ r_2 &: \quad x_2 + 2x_3 = 3, \\ r_3 &: \quad 0 = 0. \end{aligned}$$

Since the third row corresponds to the equation $0 = 0$, which is always true, the system is **consistent**.

Step 7: Express the solution using a free variable

$$\begin{aligned} x_1 &= -2 + x_3, \\ x_2 &= 3 - 2x_3, \\ x_3 &= x_3. \end{aligned}$$

◀

Chapter 8 Vectors and Matrices

A **vector** is a mathematical object that has both **magnitude** (length) and **direction**. In two or three dimensions, a vector is often represented as an arrow pointing from one point to another. For example, the vector $\boldsymbol{v} = (x_1, x_2)$ in 2D space describes a movement from the origin $(0, 0)$ to the point (x_1, x_2).

Vectors are used to represent quantities like velocity, force, and displacement, which require both magnitude and direction to fully describe them. Vectors can be added together, scaled by a number, and decomposed into components.

Matrices, on the other hand, are rectangular arrays of numbers arranged in rows and columns. They are used to represent and solve systems of linear equations, perform transformations, and encode relationships between sets of vectors.

8.1 Vectors in $\mathbb{R}^2$ and $\mathbb{R}^n$

In linear algebra, vectors are often represented as **column matrices**. For example, the vector $\boldsymbol{v} = (x_1, x_2)$ can be written as a column matrix:

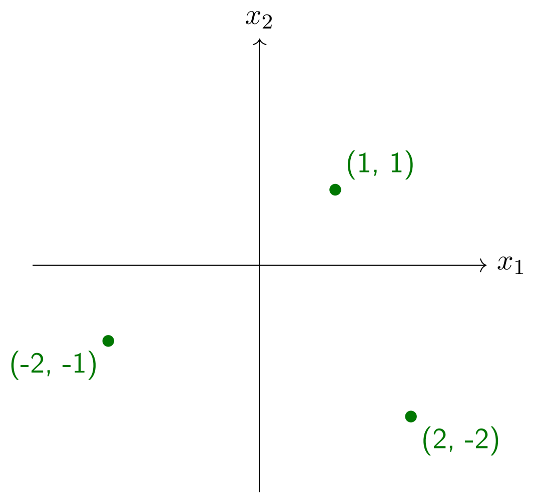
$$\boldsymbol{v} = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$$

We say that two vectors are **equal** if and only if their corresponding entries are equal.

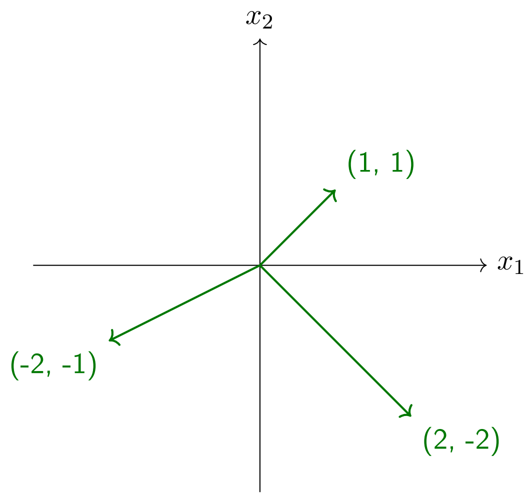
Example 8.1 The following are vectors in $\mathbb{R}^2$ (a.k.a. the plane consisting of ordered pairs of real numbers):

$$\boldsymbol{u} = \begin{bmatrix} 3 \\ 5 \end{bmatrix}, \quad \boldsymbol{v} = \begin{bmatrix} 5 \\ 3 \end{bmatrix}$$

They are not equal because their corresponding entries do not match.



(a) Points in the plane



(b) Vectors from the origin

Figure 8.1: Points and vectors in the plane

The sum of two vectors $\boldsymbol{u}$ and $\boldsymbol{v}$ in $\mathbb{R}^2$, denoted $\boldsymbol{u} + \boldsymbol{v}$, is obtained by adding the corresponding entries of

$\boldsymbol{u}$ and $\boldsymbol{v}$. Given a real number c, the scalar multiple of $\boldsymbol{u}$ by c, denoted $c\boldsymbol{u}$, is obtained by multiplying each entry in $\boldsymbol{u}$ by c.

Example 8.2 If $\boldsymbol{u}$ and $\boldsymbol{v}$ are as in the preceding example, then

$$\boldsymbol{u}+\boldsymbol{v} = \begin{bmatrix} 3 \\ 5 \end{bmatrix} + \begin{bmatrix} 5 \\ 3 \end{bmatrix} = \begin{bmatrix} 3+5 \\ 5+3 \end{bmatrix} = \begin{bmatrix} 8 \\ 8 \end{bmatrix}, \text{ and}$$
$$6\boldsymbol{u} = 6\begin{bmatrix} 3 \\ 5 \end{bmatrix} = \begin{bmatrix} 6\cdot 3 \\ 6\cdot 5 \end{bmatrix} = \begin{bmatrix} 18 \\ 30 \end{bmatrix}$$

Remark: It is often helpful to identify a vector $\begin{bmatrix} a \\ b \end{bmatrix} \in \mathbb{R}^2$ with a geometric point (a, b) in the plane in order to get a picture of what we are working with.Please see figure 8.1 for an example.

If $\boldsymbol{u}$ and $\boldsymbol{v}$ in $\mathbb{R}^2$ are thought of as points in the plane, then $\boldsymbol{u}+\boldsymbol{v}$ corresponds to the fourth vertex of the parallelogram whose other vertices are $\mathbf{0}, \boldsymbol{u}$, and $\boldsymbol{v}$. Note: By $\mathbf{0}$, we mean the zero vector, or the vector whose entries are all zero. In $\mathbb{R}^2$, we have $\mathbf{0} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$.

We call this the *parallelogram law of addition* and it can be seen in figure 8.2.

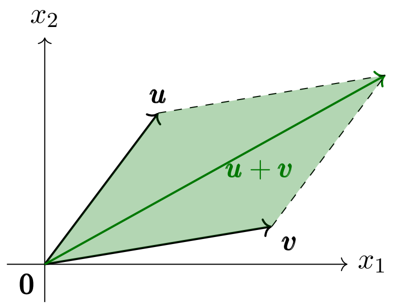

x_2
$\boldsymbol{u}+\boldsymbol{v}$
$\boldsymbol{u}$
$\boldsymbol{u}+\boldsymbol{v}$
$\boldsymbol{v}$
x_1
$\mathbf{0}$

Figure 8.2: The parallelogram law of vector addition in $\mathbb{R}^2$.

These ideas generalise to higher-dimensional spaces. More specifically, we can define $\mathbb{R}^n$ as follows.

Definition 8.1

For each positive integer n, we let $\mathbb{R}^n$ denote the collection of ordered n-tuples with each entry in $\mathbb{R}$. We often write these elements as $n \times 1$ matrices. We define addition and scalar multiplication of vectors in $\mathbb{R}^n$ in the same way as we do for $\mathbb{R}^2$. That is, we go coordinate-by-coordinate. ♣

Example 8.3 If $u_1, u_2, \ldots, u_n \in \mathbb{R}$, then

$$\boldsymbol{u} = \begin{bmatrix} u_1 \\ u_2 \\ \vdots \\ u_n \end{bmatrix} \in \mathbb{R}^n.$$

Example 8.4 If $\boldsymbol{u}$ and $\boldsymbol{v}$ are in $\mathbb{R}^n$ (with entries denoted $u_1, \ldots, u_n$ and $v_1, \ldots, v_n$, respectively), and $c \in \mathbb{R}$, then

$$\boldsymbol{u}+\boldsymbol{v} = \begin{bmatrix} u_1 \\ u_2 \\ \vdots \\ u_n \end{bmatrix} + \begin{bmatrix} v_1 \\ v_2 \\ \vdots \\ v_n \end{bmatrix} = \begin{bmatrix} u_1+v_1 \\ u_2+v_2 \\ \vdots \\ u_n+v_n \end{bmatrix}, \text{ and } \quad c\boldsymbol{u} = c\begin{bmatrix} u_1 \\ u_2 \\ \vdots \\ u_n \end{bmatrix} = \begin{bmatrix} cu_1 \\ cu_2 \\ \vdots \\ cu_n \end{bmatrix}$$

Algebraic Properties of $\mathbb{R}^n$

Let $\boldsymbol{u}, \boldsymbol{v}, \boldsymbol{w} \in \mathbb{R}^n$ and $c, d \in \mathbb{R}$. Then the following properties hold:

1. **Commutative Property of Addition:** $\boldsymbol{u}+\boldsymbol{v} = \boldsymbol{v}+\boldsymbol{u}$.
2. **Associative Property of Addition:** $(\boldsymbol{u}+\boldsymbol{v})+\boldsymbol{w} = \boldsymbol{u}+(\boldsymbol{v}+\boldsymbol{w})$.
3. **Additive Identity:** There exists a vector $\mathbf{0} \in \mathbb{R}^n$ such that $\boldsymbol{u}+\mathbf{0} = \boldsymbol{u}$ for all $\boldsymbol{u} \in \mathbb{R}^n$.
4. **Additive Inverse:** For each $\boldsymbol{u} \in \mathbb{R}^n$, there exists a vector $-\boldsymbol{u} \in \mathbb{R}^n$ such that $\boldsymbol{u}+(-\boldsymbol{u}) = \mathbf{0}$.
5. **Distributive Property:** $c(\boldsymbol{u}+\boldsymbol{v}) = c\boldsymbol{u}+c\boldsymbol{v}$ and $(c+d)\boldsymbol{u} = c\boldsymbol{u}+d\boldsymbol{u}$.
6. **Associative Property of Scalar Multiplication:** $c(d\boldsymbol{u}) = (cd)\boldsymbol{u}$.
7. **Multiplicative Identity:** $1\boldsymbol{u} = \boldsymbol{u}$.

A **linear combination** is a way to combine vectors using scalar multiplication and addition. Given a set of vectors, we multiply each by a scalar and then sum the results. Linear combinations help us understand how vectors relate to each other and whether one vector can be expressed in terms of others.

Definition 8.2 (Linear Combinations)

Given a set of vectors $\boldsymbol{v}_1, \boldsymbol{v}_2, \ldots, \boldsymbol{v}_p \in \mathbb{R}^n$ and scalars $c_1, c_2, \ldots, c_p \in \mathbb{R}$, the vector $\boldsymbol{y}$ given by

$$\boldsymbol{y} = c_1\boldsymbol{v}_1 + \cdots + c_p\boldsymbol{v}_p$$

is called a linear combination of $\boldsymbol{v}_1, \boldsymbol{v}_2, \ldots, \boldsymbol{v}_p$ with weights $c_1, c_2, \ldots, c_p$. ♣

Example 8.5 Let $\boldsymbol{v}_1 = \begin{bmatrix} 1 \\ 2 \end{bmatrix}$ and $\boldsymbol{v}_2 = \begin{bmatrix} 1 \\ -1 \end{bmatrix}$. Some linear combinations of $\boldsymbol{v}_1$ and $\boldsymbol{v}_2$ include

$$\begin{aligned} \mathbf{0} &= 0\boldsymbol{v}_1 + 0\boldsymbol{v}_2 \\ \begin{bmatrix} 3 \\ 0 \end{bmatrix} &= \boldsymbol{v}_1 + 2\boldsymbol{v}_2 \\ \begin{bmatrix} -5 \\ -1 \end{bmatrix} &= -2\boldsymbol{v}_1 - 3\boldsymbol{v}_2, \text{ and} \\ \begin{bmatrix} 5 \\ 1 \end{bmatrix} &= 2\boldsymbol{v}_1 + 3\boldsymbol{v}_2 \end{aligned}$$

Vector Equations and Linear Systems

It is often the case that we wish to know if some vector $\boldsymbol{b}$ can be formed as a linear combination of some other set of vectors $\boldsymbol{a}_1, \ldots, \boldsymbol{a}_n$. The process for figuring this out is given by the following.

Using Matrices to Determine Linear Combinations

A vector equation

$$x_1\boldsymbol{a}_1 + x_2\boldsymbol{a}_2 + \cdots + x_n\boldsymbol{a}_n = \boldsymbol{b}$$

has the same solution set as the linear system whose augmented matrix is

$$\begin{bmatrix} \boldsymbol{a}_1 & \boldsymbol{a}_2 & \ldots & \boldsymbol{a}_n & \boldsymbol{b} \end{bmatrix}.$$

More specifically, if the $\boldsymbol{a}_i$'s are in $\mathbb{R}^m$ with

$$\boldsymbol{a}_1 = \begin{bmatrix} a_{11} \\ a_{21} \\ \vdots \\ a_{m1} \end{bmatrix}, \quad \boldsymbol{a}_2 = \begin{bmatrix} a_{12} \\ a_{22} \\ \vdots \\ a_{m2} \end{bmatrix}, \ldots, \quad \boldsymbol{a}_n = \begin{bmatrix} a_{1n} \\ a_{2n} \\ \vdots \\ a_{mn} \end{bmatrix}, \quad \text{and} \quad \boldsymbol{b} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_m \end{bmatrix}$$

then you would row reduce the matrix

$$\begin{array}{ccccc} \left[\begin{matrix} a_{11} & a_{12} & \ldots & a_{1n} & b_1 \\ a_{21} & a_{22} & \ldots & a_{2n} & b_2 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ a_{m1} & a_{m2} & \ldots & a_{mn} & b_m \end{matrix}\right] \\ \begin{matrix} \uparrow & \uparrow & \ldots & \uparrow & \uparrow \\ \boldsymbol{a}_1 & \boldsymbol{a}_2 & \ldots & \boldsymbol{a}_n & \boldsymbol{b} \end{matrix} \end{array}$$

to determine if there is some set of weights $x_1, \ldots, x_n$ that work.

Example 8.6 Let

$$\boldsymbol{a}_1 = \begin{bmatrix} 2 \\ -1 \\ 1 \end{bmatrix}, \quad \boldsymbol{a}_2 = \begin{bmatrix} 0 \\ 8 \\ -2 \end{bmatrix}, \quad \boldsymbol{a}_3 = \begin{bmatrix} 6 \\ 5 \\ 1 \end{bmatrix}, \quad \text{and } \boldsymbol{b} = \begin{bmatrix} 10 \\ 3 \\ 7 \end{bmatrix}$$

We determine if $\boldsymbol{b}$ is a linear combination of $\boldsymbol{a}_1, \boldsymbol{a}_2, \boldsymbol{a}_3$, i.e. if there is some set of weights x_1, x_2, x_3 such that $x_1\boldsymbol{a}_1 + x_2\boldsymbol{a}_2 + x_3\boldsymbol{a}_3 = \boldsymbol{b}$. By the above, we translate this question to the matrix setting.

$$\begin{bmatrix} 2 & 0 & 6 & 10 \\ -1 & 8 & 5 & 3 \\ 1 & -2 & 1 & 7 \end{bmatrix} \quad r_1 \leftrightarrow r_3 \quad \begin{bmatrix} 1 & -2 & 1 & 7 \\ -1 & 8 & 5 & 3 \\ 2 & 0 & 6 & 10 \end{bmatrix}$$

$$\begin{matrix} r_2 \mapsto r_2 + r_1 \\ r_3 \mapsto r_3 - 2r_1 \end{matrix} \quad \begin{bmatrix} 1 & -2 & 1 & 7 \\ 0 & 6 & 6 & 10 \\ 0 & 4 & 4 & -4 \end{bmatrix}$$

$$r_3 \mapsto r_3 - \frac{4}{6}r_2 \quad \begin{bmatrix} 1 & -2 & 1 & 7 \\ 0 & 6 & 6 & 10 \\ 0 & 0 & 0 & \blacksquare \end{bmatrix}.$$

Since the symbol $\blacksquare$ denotes something nonzero, we see that there's no solution, i.e. that $\boldsymbol{b}$ is not a linear combination of $\boldsymbol{a}_1, \boldsymbol{a}_2, \boldsymbol{a}_3$.

Example 8.7 Let

$$\boldsymbol{a}_1 = \begin{bmatrix} 1 \\ 0 \\ 1 \end{bmatrix}, \quad \boldsymbol{a}_2 = \begin{bmatrix} -4 \\ 6 \\ -4 \end{bmatrix}, \quad \boldsymbol{a}_3 = \begin{bmatrix} -6 \\ 7 \\ 5 \end{bmatrix}, \quad \text{and } \boldsymbol{b} = \begin{bmatrix} 11 \\ -5 \\ 9 \end{bmatrix}$$

We determine if $\boldsymbol{b}$ is a linear combination of $\boldsymbol{a}_1, \boldsymbol{a}_2, \boldsymbol{a}_3$. We reduce the corresponding matrix to echelon form:

$$\begin{bmatrix} 1 & -4 & -6 & 11 \\ 0 & 6 & 7 & -5 \\ 1 & -4 & 5 & 9 \end{bmatrix} \quad r_3 \mapsto r_3 - r_1 \quad \begin{bmatrix} 1 & -4 & -6 & 11 \\ 0 & 6 & 7 & -5 \\ 0 & 0 & 11 & -2 \end{bmatrix}$$

and we see that there is a solution. Now we find what weights x_1, x_2, x_3 work by finding the reduced echelon form.

$$r_3 \mapsto \frac{1}{11} r_3 \qquad \begin{bmatrix} 1 & -4 & -6 & 11 \\ 0 & 6 & 7 & -5 \\ 0 & 0 & 1 & -\frac{2}{11} \end{bmatrix}$$

$$\begin{matrix} r_2 \mapsto r_2 - 7r_3 \\ r_1 \mapsto r_1 + 6r_3 \end{matrix} \qquad \begin{bmatrix} 1 & -4 & 0 & \frac{109}{11} \\ 0 & 6 & 0 & -\frac{41}{11} \\ 0 & 0 & 1 & -\frac{2}{11} \end{bmatrix}$$

$$r_2 \mapsto \frac{1}{6} r_2 \qquad \begin{bmatrix} 1 & -4 & 0 & \frac{109}{11} \\ 0 & 1 & 0 & -\frac{41}{66} \\ 0 & 0 & 1 & -\frac{2}{11} \end{bmatrix}$$

$$r_1 \mapsto r_1 + 4r_2 \qquad \begin{bmatrix} 1 & 0 & 0 & \frac{245}{38} \\ 0 & 1 & 0 & -\frac{41}{66} \\ 0 & 0 & 1 & -\frac{2}{11} \end{bmatrix},$$

so $(x_1, x_2, x_3) = \left(\frac{245}{33}, -\frac{41}{66}, -\frac{2}{11}\right)$. Since there are no free variables, this is the unique solution.

The **span** of a set of vectors is the collection of all possible linear combinations of those vectors. In other words, it's the set of all vectors you can reach by scaling and adding the given vectors. The span gives us insight into the "space" those vectors cover:

Definition 8.3 (Span of a Set of Vectors)

If $\boldsymbol{v}_1, \ldots, \boldsymbol{v}_p$ are in $\mathbb{R}^n$, then the set of all linear combinations of $\boldsymbol{v}_1, \ldots, \boldsymbol{v}_p$ is denoted by Span $\{\boldsymbol{v}_1, \ldots, \boldsymbol{v}_p\}$ and is called the subset of $\mathbb{R}^n$ spanned by $\boldsymbol{v}_1, \ldots, \boldsymbol{v}_p$. In other words, the span of $\boldsymbol{v}_1, \ldots, \boldsymbol{v}_p$ is all vectors that can be written in the form

$$c_1 \boldsymbol{v}_1 + \cdots + c_p \boldsymbol{v}_p$$

with $c_1, \ldots, c_p$ scalars.

♣

8.2 Matrix Equations

In the previous sections, we explored different ways of representing linear relationships. For instance, we looked at individual linear equations, such as

$$2x_1 + 3x_2 = 5,$$

and systems of linear equations, like

$$\begin{aligned} x_1 + 2x_2 &= 4 \\ 3x_1 - x_2 &= 2. \end{aligned}$$

We also expressed these systems as vector equations, such as

$$x_1 \begin{bmatrix} 1 \\ 3 \end{bmatrix} + x_2 \begin{bmatrix} 2 \\ -1 \end{bmatrix} = \begin{bmatrix} 4 \\ 2 \end{bmatrix}.$$

Now, it's time to introduce a powerful new form: the **matrix equation**. This compact representation allows us to handle systems of linear equations efficiently using matrix notation. In this form, our system becomes

$$A\boldsymbol{x} = \boldsymbol{b},$$

where A is a matrix, $\boldsymbol{x}$ is a vector of variables, and $\boldsymbol{b}$ is the result vector. Matrix equations give us a structured, algebraic approach to solving systems.

Definition 8.4 (Matrix Equation)

If A is an $m \times n$ matrix with columns $\boldsymbol{a}_1, \ldots, \boldsymbol{a}_n$, and if $\boldsymbol{x} \in \mathbb{R}^n$, then the product of A and $\boldsymbol{x}$, denoted by $A\boldsymbol{x}$, is

$$A\boldsymbol{x} = \begin{bmatrix} \boldsymbol{a}_1 & \boldsymbol{a}_2 & \ldots & \boldsymbol{a}_n \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} = x_1\boldsymbol{a}_1 + x_2\boldsymbol{a}_2 + \cdots + x_n\boldsymbol{a}_n$$

♣

Example 8.8

$$\begin{aligned} \begin{bmatrix} 4 & 1 & 2 \\ 8 & 0 & 3 \end{bmatrix} \begin{bmatrix} 2 \\ 1 \\ 4 \end{bmatrix} &= 2\begin{bmatrix} 4 \\ 8 \end{bmatrix} + 1\begin{bmatrix} 1 \\ 0 \end{bmatrix} + 4\begin{bmatrix} 2 \\ 3 \end{bmatrix} \\ &= \begin{bmatrix} 8 \\ 16 \end{bmatrix} + \begin{bmatrix} 1 \\ 0 \end{bmatrix} + \begin{bmatrix} 8 \\ 12 \end{bmatrix} \\ &= \begin{bmatrix} 17 \\ 28 \end{bmatrix} \end{aligned}$$

Building a bit on the main result from the preceding section, we have the following theorem.

Theorem 8.1

If A is an $m \times n$ matrix, with columns $\boldsymbol{a}_1, \ldots, \boldsymbol{a}_n \in \mathbb{R}^m$ and if $\boldsymbol{b} \in \mathbb{R}^m$, the matrix equation

$$A\boldsymbol{x} = \boldsymbol{b}$$

has the same solution set as the vector equation

$$x_1\boldsymbol{a}_1 + x_2\boldsymbol{a}_2 + \cdots + x_n\boldsymbol{a}_n = \boldsymbol{b}$$

which, in turn, has the same solution set as the system of linear equations with augmented matrix

$$\begin{bmatrix} \boldsymbol{a}_1 & \boldsymbol{a}_2 & \ldots & \boldsymbol{a}_n & \boldsymbol{b} \end{bmatrix}.$$

♡

This theorem is essential because it ties together the three forms of representing and solving systems of linear equations: matrix equations, vector equations, and systems of equations with augmented matrices. It shows that no matter which form we use, they all lead to the same solution set.

By stating that the matrix equation $A\boldsymbol{x} = \boldsymbol{b}$ is equivalent to the vector equation $x_1\boldsymbol{a}_1+x_2\boldsymbol{a}_2+\cdots+x_n\boldsymbol{a}_n = \boldsymbol{b}$, we can interpret solving a matrix equation as finding the right combination of the columns of A that yields $\boldsymbol{b}$. This connection allows us to visualize the problem in terms of vector spaces and linear combinations.

Moreover, the theorem shows that this same process is reflected in the augmented matrix, where row reduction reveals the solution through elementary row operations. So whether we approach the problem algebraically, geometrically, or algorithmically, we are dealing with the same underlying structure. This unifying perspective simplifies our approach to solving systems and provides flexibility in how we choose to represent and manipulate the problem.

Remark: The equation $A\boldsymbol{x} = \boldsymbol{b}$ has a solution if and only if $\boldsymbol{b}$ is a linear combination of the columns of A.

The following theorem tells us when the column vectors of an $m \times n$ matrix (the columns are vectors in $\mathbb{R}^m$, since there are m rows) can be used to generate all of $\mathbb{R}^m$. This basically summarises several things we have already seen in different contexts.

Theorem 8.2

Let A be an $m \times n$ matrix. Then the following statements are equivalent:

(a) For each $\boldsymbol{b} \in \mathbb{R}^m$, the equation $A\boldsymbol{x} = \boldsymbol{b}$ has a solution.
(b) Each $\boldsymbol{b} \in \mathbb{R}^m$ is a linear combination of the columns of A.
(c) The columns of A span $\mathbb{R}^m$.
(d) The matrix A has a pivot position in every row.

♡

This theorem shows that several seemingly different ideas are in fact equivalent. The existence of a solution to the matrix equation $A\boldsymbol{x} = \boldsymbol{b}$ (statement (a)) depends on whether the columns of A can form any vector in $\mathbb{R}^m$ (statement (b)). Geometrically, this means the columns span the entire space $\mathbb{R}^m$ (statement (c)).

Finally, the presence of a pivot position in every row (statement (d)) gives an algebraic condition that guarantees the span of the columns covers $\mathbb{R}^m$, ensuring that there is always a solution to $A\boldsymbol{x} = \boldsymbol{b}$.

This equivalence is a powerful tool, as it connects solutions, linear combinations, and geometric interpretations, while also providing a concrete method (checking for pivot positions) to verify these properties.

Row-Vector Rule for Computing $A\boldsymbol{x}$

Assuming the product $A\boldsymbol{x}$ is defined, the i-th entry in $A\boldsymbol{x}$ is the sum of the products of corresponding entries from row i of A and the vector $\boldsymbol{x}$. In other words, the i-th entry of $A\boldsymbol{x}$ is the dot product of the vector forming the i-th row of A and the vector $\boldsymbol{x}$.

This rule simplifies the matrix multiplication process by breaking it down into smaller, familiar operations — dot products — making it easier to compute and understand. It also highlights the connection between matrix multiplication and linear combinations, as the product $A\boldsymbol{x}$ is a linear combination of the rows of A with weights given by the entries of $\boldsymbol{x}$.

Example 8.9 Let

$$A = \begin{bmatrix} 1 & -1 & 4 \\ 5 & 0 & 2 \end{bmatrix}, \text{ and } \boldsymbol{x} = \begin{bmatrix} 2 \\ 1 \\ 5 \end{bmatrix}$$

Then

$$\begin{aligned} \begin{bmatrix} 1 & -1 & 4 \\ 5 & 0 & 2 \end{bmatrix} \begin{bmatrix} 2 \\ 1 \\ 5 \end{bmatrix} &= \begin{bmatrix} 1 \cdot 2 + (-1) \cdot 1 + 4 \cdot 5 \\ 5 \cdot 2 + 0 \cdot 1 + 2 \cdot 5 \end{bmatrix} \\ &= \begin{bmatrix} 21 \\ 20 \end{bmatrix} \end{aligned}$$

Notice that the number of columns in A must match the number of rows in $\boldsymbol{x}$, for otherwise the dot product would not make sense!

The next theorem captures two important properties of matrix-vector multiplication: **distributivity** and **scalar multiplication**. These properties mirror the familiar rules of algebra but now apply in the context of matrices and vectors.

Theorem 8.3

If A is an $m \times n$ matrix, $\boldsymbol{u}$ and $\boldsymbol{v}$ are vectors in $\mathbb{R}^n$, and c is a scalar, then:

(a) $A(\boldsymbol{u} + \boldsymbol{v}) = A\boldsymbol{u} + A\boldsymbol{v}$

(b) $A(c\boldsymbol{u}) = c(A\boldsymbol{u})$

♡

Property (a) shows that matrix multiplication distributes over vector addition. It means that multiplying A by the sum of two vectors is the same as multiplying A by each vector separately and then adding the results. Property (b) illustrates that scalar multiplication commutes with matrix multiplication. Multiplying a vector by a scalar first, then applying the matrix, gives the same result as applying the matrix first and then multiplying the resulting vector by the scalar.

Example 8.10 Let

$$A = \begin{bmatrix} 1 & 5 & -2 & 0 \\ -3 & 1 & 9 & -5 \\ 4 & -8 & -1 & 7 \end{bmatrix}, \quad \boldsymbol{p} = \begin{bmatrix} 3 \\ -2 \\ 0 \\ -4 \end{bmatrix}, \quad \boldsymbol{b} = \begin{bmatrix} -7 \\ 9 \\ 0 \end{bmatrix}$$

It can be shown that $\boldsymbol{p}$ is a solution of $A\boldsymbol{x} = \boldsymbol{b}$. Use this fact to write $\boldsymbol{b}$ as a linear combination of the columns of A.

Solution: Since $\boldsymbol{p}$ is a solution to $A\boldsymbol{x} = \boldsymbol{b}$, we have:

$$A\boldsymbol{p} = \boldsymbol{b}.$$

We can express $\boldsymbol{b}$ as a linear combination of the columns of A using the entries of $\boldsymbol{p}$. Let $\boldsymbol{a}_1$, $\boldsymbol{a}_2$, $\boldsymbol{a}_3$, and $\boldsymbol{a}_4$ denote the columns of A, so:

$$A = \left[\boldsymbol{a}_1 \mid \boldsymbol{a}_2 \mid \boldsymbol{a}_3 \mid \boldsymbol{a}_4\right].$$

Thus,

$$\boldsymbol{b} = p_1\boldsymbol{a}_1 + p_2\boldsymbol{a}_2 + p_3\boldsymbol{a}_3 + p_4\boldsymbol{a}_4.$$

Now, we compute each term one at a time.

Compute $p_1\boldsymbol{a}_1$:

$$p_1\boldsymbol{a}_1 = 3\begin{bmatrix} 1 \\ -3 \\ 4 \end{bmatrix} = \begin{bmatrix} 3 \\ -9 \\ 12 \end{bmatrix}.$$

Compute $p_2\boldsymbol{a}_2$:

$$p_2\boldsymbol{a}_2 = (-2)\begin{bmatrix} 5 \\ 1 \\ -8 \end{bmatrix} = \begin{bmatrix} -10 \\ -2 \\ 16 \end{bmatrix}.$$

Compute $p_3\boldsymbol{a}_3$:

$$p_3\boldsymbol{a}_3 = 0\begin{bmatrix} -2 \\ 9 \\ -1 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}.$$

Compute $p_4\boldsymbol{a}_4$:

$$p_4\boldsymbol{a}_4 = (-4)\begin{bmatrix} 0 \\ -5 \\ 7 \end{bmatrix} = \begin{bmatrix} 0 \\ 20 \\ -28 \end{bmatrix}.$$

Add the computed vectors to find $\boldsymbol{b}$:

$$\boldsymbol{b} = p_1\boldsymbol{a}_1 + p_2\boldsymbol{a}_2 + p_3\boldsymbol{a}_3 + p_4\boldsymbol{a}_4 = \begin{bmatrix} 3 \\ -9 \\ 12 \end{bmatrix} + \begin{bmatrix} -10 \\ -2 \\ 16 \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} + \begin{bmatrix} 0 \\ 20 \\ -28 \end{bmatrix}.$$

Now, compute the sum step by step.

Step 1: Add $p_1\boldsymbol{a}_1$ and $p_2\boldsymbol{a}_2$:

$$\begin{bmatrix} 3 \\ -9 \\ 12 \end{bmatrix} + \begin{bmatrix} -10 \\ -2 \\ 16 \end{bmatrix} = \begin{bmatrix} -7 \\ -11 \\ 28 \end{bmatrix}.$$

Step 2: Add $p_3\boldsymbol{a}_3$ (which is zero) to the result:

$$\begin{bmatrix} -7 \\ -11 \\ 28 \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} -7 \\ -11 \\ 28 \end{bmatrix}.$$

Step 3: Add $p_4\boldsymbol{a}_4$ to the result:

$$\begin{bmatrix} -7 \\ -11 \\ 28 \end{bmatrix} + \begin{bmatrix} 0 \\ 20 \\ -28 \end{bmatrix} = \begin{bmatrix} -7 \\ 9 \\ 0 \end{bmatrix}.$$

Final Result:

$$\boldsymbol{b} = \begin{bmatrix} -7 \\ 9 \\ 0 \end{bmatrix}.$$

Conclusion:

We have expressed $\boldsymbol{b}$ as a linear combination of the columns of A using the entries of $\boldsymbol{p}$:

$$\boldsymbol{b} = 3\boldsymbol{a}_1 - 2\boldsymbol{a}_2 + 0\boldsymbol{a}_3 - 4\boldsymbol{a}_4.$$

◀

8.3 Solution Sets of Linear Systems

Solution sets of linear systems play a crucial role in the study of linear algebra and will reappear in various contexts throughout the subject. In this section, we use vector notation to provide clear, explicit, and geometric descriptions of these solution sets.

A linear system is said to be *homogeneous* if it can be written in the form

$$A\boldsymbol{x} = \mathbf{0}$$

where A is an $m \times n$ matrix and $\mathbf{0}$ is the zero vector in $\mathbb{R}^m$. Such a system always has the *trivial solution*, namely $\boldsymbol{x} = \mathbf{0}$ (the zero vector in $\mathbb{R}^n$, not $\mathbb{R}^m$).

Since homogeneous systems always have the trivial solution, the interesting question is whether or not they have *nontrivial solutions*.

Remark: The homogeneous equation $A\boldsymbol{x} = \mathbf{0}$ has a nontrivial solution if and only if the equation has at least one free variable.

Example 8.11 Determine if the following homogeneous system has a nontrivial solution. If it does, describe the solution set using the free variable(s).

$$\begin{aligned} 2x_1 + x_2 - 3x_3 &= 0 \\ x_1 - x_2 + x_3 &= 0 \\ -2x_1 + 5x_2 - 7x_3 &= 0 \end{aligned}$$

Solution: We let A be the coefficient matrix of the system, and reduce the augmented mat rix $\begin{bmatrix} A & 0 \end{bmatrix}$ to echelon form (notice how we put the second equation first in order to simplify the computation a bit):

$$\begin{bmatrix} 1 & -1 & 1 & 0 \\ 2 & 1 & -3 & 0 \\ -2 & 5 & -7 & 0 \end{bmatrix} \sim \begin{bmatrix} 1 & -1 & 1 & 0 \\ 0 & 3 & -5 & 0 \\ 0 & 3 & -5 & 0 \end{bmatrix}$$
$$\sim \begin{bmatrix} 1 & -1 & 1 & 0 \\ 0 & 3 & -5 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

so we see that x_3 is a free variable, and as a result we have nontrivial solutions. Now we get the reduced echelon form:

$$\begin{bmatrix} 1 & -1 & 1 & 0 \\ 0 & 3 & -5 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \sim \begin{bmatrix} 1 & -1 & 1 & 0 \\ 0 & 1 & -\frac{5}{3} & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$
$$\sim \begin{bmatrix} 1 & 0 & -\frac{2}{3} & 0 \\ 0 & 1 & -\frac{5}{3} & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

This gives us the equations

$$x_1 - \frac{2}{3}x_3 = 0 \quad \Longrightarrow x_1 = \frac{2}{3}x_3$$
$$x_2 - \frac{5}{3}x_3 = 0 \quad \Longrightarrow x_2 = \frac{5}{3}x_3$$

In vector form, the solution $\boldsymbol{x}$ of the equation $A\boldsymbol{x} = 0$ is written

$$\boldsymbol{x} = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} \frac{2}{3}x_3 \\ \frac{5}{3}x_3 \\ x_3 \end{bmatrix} = x_3 \begin{bmatrix} \frac{2}{3} \\ \frac{5}{3} \\ 1 \end{bmatrix}$$

Since x_3 can be anything, in geometric terms this solution set describes the line in $\mathbb{R}^3$ extending from the origin through the point $\left(\frac{2}{3}, \frac{5}{3}, 1\right)$. ◀

Parametric Vector Form

Whenever a solution set is described explicitly with vectors (as in the preceding example), we say that the solution is in **parametric vector form**.

Example 8.12 Describe all solutions of the homogeneous equation

$$x_1 - 2x_2 - 5x_3 = 0$$

Solution:

Writing this system in a matrix, we would see that x_2 and x_3 are free variables, and x_1 is a basic variable with $x_1 = 2x_2 + 5x_3$. Hence the general solution is

$$\boldsymbol{x} = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 2x_2 + 5x_3 \\ x_2 \\ x_3 \end{bmatrix} = x_2 \begin{bmatrix} 2 \\ 1 \\ 0 \end{bmatrix} + x_3 \begin{bmatrix} 5 \\ 0 \\ 1 \end{bmatrix}$$

where the right-most part of this equation is the **parametric vector** form of the solution set. In this case, the entire solution set is a plane in $\mathbb{R}^3$ that contains the two lines extending from $(0,0,0)$ to $(2,1,0)$, and from $(0,0,0)$ to $(5,0,1)$. ◀

When a *nonhomogeneous* linear system has many solutions, the general solution can be written in parametric vector form as one vector plus arbitrary linear combinations of vectors that satisfy the corresponding homogeneous system.

Example 8.13 We reconsider the homogeneous system at the beginning of this section, except this time it will be nonhomogeneous (i.e., the right side will not be all zeroes):

$$\begin{aligned} 2x_1 + x_2 - 3x_3 &= 2 \\ x_1 - x_2 + x_3 &= 1 \\ -2x_1 + 5x_2 - 7x_3 &= -2 \end{aligned}$$

As before, we perform row operations on the augmented matrix $\begin{bmatrix} A & \boldsymbol{b} \end{bmatrix}$ where A is the same coefficient matrix and

$$\boldsymbol{b} = \begin{bmatrix} 2 \\ 1 \\ -2 \end{bmatrix}$$

and we find that

$$\begin{bmatrix} 1 & -1 & 1 & 1 \\ 2 & 1 & -3 & 2 \\ -2 & 5 & -7 & -2 \end{bmatrix} \sim \begin{bmatrix} 1 & 0 & -\frac{2}{3} & 1 \\ 0 & 1 & -\frac{5}{3} & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

Hence,

$$x_1 = 1 + \frac{2}{3}x_3, \text{ and}$$
$$x_2 = \frac{5}{3}x_3$$

This can be written as

$$\boldsymbol{x} = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 1 + \frac{2}{3}x_3 \\ \frac{5}{3}x_3 \\ x_3 \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} + x_3 \begin{bmatrix} \frac{2}{3} \\ \frac{5}{3} \\ 1 \end{bmatrix}$$

which has the form we claimed. This is the equation of the line through the point $(1,0,0)$ that is parallel to the line extending from $(0,0,0)$ through $\left(\frac{2}{3}, \frac{5}{3}, 1\right)$.

We summarise the general situation with the following theorem. It might be best to think of the conclusion as shifting all homogeneous solutions by the vector $\boldsymbol{p}$.

Theorem 8.4

Suppose $A\boldsymbol{x} = \boldsymbol{b}$ is consistent for some $\boldsymbol{b}$, and let $\boldsymbol{p}$ be a solution. Then the solution set of $A\boldsymbol{x} = \boldsymbol{b}$ is the set of all vectors of the form

$$\boldsymbol{w} = \boldsymbol{p} + \boldsymbol{v}_h,$$

where $\boldsymbol{v}_h$ is any solution of the homogeneous equation $A\boldsymbol{x} = \boldsymbol{0}$. ♡

Example 8.14 Write the general solution of

$$x_1 - 2x_2 - 5x_3 = 3$$

in parametric vector form. In geometric terms, what does this solution set look like in comparison to the solution set of the equation $x_1 - 2x_2 - 5x_3 = 0$ that we saw earlier?

Solution: Let us solve the equation:

$$x_1 - 2x_2 - 5x_3 = 3$$

To express the general solution, we solve for x_1 in terms of x_2 and x_3:

$$x_1 = 3 + 2x_2 + 5x_3,$$

$$x_2 = x_2,$$

$$x_3 = x_3.$$

We can express the solution set in *parametric vector form*:

$$\begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 3 \\ 0 \\ 0 \end{bmatrix} + x_2 \begin{bmatrix} 2 \\ 1 \\ 0 \end{bmatrix} + x_3 \begin{bmatrix} 5 \\ 0 \\ 1 \end{bmatrix}, \quad \text{for all } s, t \in \mathbb{R}.$$

Geometric Interpretation: The solution set represents a plane in $\mathbb{R}^3$. This plane is parallel to the plane defined by the homogeneous equation:

$$x_1 - 2x_2 - 5x_3 = 0$$

The homogeneous solution set can be expressed as:

$$\begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = x_2 \begin{bmatrix} 2 \\ 1 \\ 0 \end{bmatrix} + x_3 \begin{bmatrix} 5 \\ 0 \\ 1 \end{bmatrix}, \quad \text{for all } s, t \in \mathbb{R}.$$

Comparing both solution sets, we observe that the original solution is the homogeneous solution shifted by the vector $\begin{bmatrix} 3 \\ 0 \\ 0 \end{bmatrix}$. Geometrically, this means the plane defined by $x_1 - 2x_2 - 5x_3 = 3$ is parallel to, but not the same as, the plane defined by $x_1 - 2x_2 - 5x_3 = 0$. The two planes are offset along the x_1-axis by 3 units. The general solution of the equation $x_1 - 2x_2 - 5x_3 = 3$ is a plane in $\mathbb{R}^3$ that is parallel to the solution set of the homogeneous equation $x_1 - 2x_2 - 5x_3 = 0$, but shifted away from the origin. ◀

Example 8.15 Find the parametric equation of the line through $\boldsymbol{a} = \begin{bmatrix} 3 \\ -2 \end{bmatrix}$ that is also parallel to $\boldsymbol{b} = \begin{bmatrix} -7 \\ 6 \end{bmatrix}$.

Solution: The line through $\boldsymbol{a}$ parallel to $\boldsymbol{b}$ is given by the equation $\boldsymbol{r} = \boldsymbol{a} + t\boldsymbol{b}$, where t is a parameter. Substituting the given vectors, we have

$$\begin{aligned}\boldsymbol{r} &= \begin{bmatrix} 3 \\ -2 \end{bmatrix} + t\begin{bmatrix} -7 \\ 6 \end{bmatrix} \\ &= \begin{bmatrix} 3-7t \\ -2+6t \end{bmatrix}\end{aligned}$$

This is the parametric equation of the line through $\boldsymbol{a}$ that is parallel to $\boldsymbol{b}$. ◀

8.4 Linear Independence

We conclude this section by introducing one of the most important concepts in linear algebra: *linear independence*. This concept is fundamental to understanding the structure of vectors and their relationships in vector spaces.

Definition 8.5

An indexed set of vectors $\{\boldsymbol{v}_1, \dots, \boldsymbol{v}_p\} \subset \mathbb{R}^n$ is said to be **linearly independent** if the vector equation

$$x_1\boldsymbol{v}_1 + \cdots + x_p\boldsymbol{v}_p = \boldsymbol{0}$$

has only the trivial solution, i.e., if the only solution is $(x_1, \dots, x_p) = (0, \dots, 0)$. Likewise, the set $\{\boldsymbol{v}_1, \dots, \boldsymbol{v}_p\}$ is said to be **linearly dependent** if there exist weights $c_1, \dots, c_p$, not all zero, such that

$$c_1\boldsymbol{v}_1 + \cdots + c_p\boldsymbol{v}_p = \boldsymbol{0}$$

We call such an equation a *linear dependence relation* when the weights are not all zero. ♣

Remark: A set of vectors cannot be both linearly independent and linearly dependent, but it must be one of them!

Determining if a set of vectors is linearly independent is tantamount to solving the matrix equation

$$A\boldsymbol{x} = 0,$$

where the columns of A are given by the vectors. The set of vectors is linearly independent if and only if the only solution is $\boldsymbol{x} = \boldsymbol{0}$. Otherwise, there is some linear dependence relation.

Example 8.16 Determine if the following vectors in $\mathbb{R}^3$ are linearly independent:

$$\boldsymbol{v}_1 = \begin{bmatrix} 2 \\ 1 \\ 2 \end{bmatrix}, \quad \boldsymbol{v}_2 = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix}, \quad \boldsymbol{v}_3 = \begin{bmatrix} 0 \\ 1 \\ 1 \end{bmatrix}$$

Solution: We begin by reducing the corresponding augmented matrix to echelon form:

$$\begin{bmatrix} 2 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 \\ 2 & 1 & 1 & 0 \end{bmatrix} \sim \begin{bmatrix} 1 & 1 & 1 & 0 \\ 2 & 1 & 0 & 0 \\ 2 & 1 & 1 & 0 \end{bmatrix}$$
$$\sim \begin{bmatrix} 1 & 1 & 1 & 0 \\ 0 & -1 & -2 & 0 \\ 0 & -1 & -1 & 0 \end{bmatrix}$$
$$\sim \begin{bmatrix} 1 & 1 & 1 & 0 \\ 0 & -1 & -2 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

from which we see that there is a solution. Now we want to know if the solution is the trivial solution $(0, 0, 0)$. Normally, we would continue row operations until we reach reduced echelon form, but we can be smarter about this. Notice first of all that there are no bad rows (so there is at least one solution), which we expect since a homogeneous system always has at least the trivial solution. Since there are no free variables, we see there is exactly one solution. This tells us that the solution must be

$$\boldsymbol{x} = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}$$

Hence the set of vectors $\{\boldsymbol{v}_1, \boldsymbol{v}_2, \boldsymbol{v}_3\}$ is linearly independent. ◀

In general, it is very useful to determine if some set of vectors is linearly independent, so it is good to have some theorems to handle this problem quickly in certain special cases.

Theorem 8.5

(a) If a set of vectors contains the zero vector, then the set is linearly dependent.

(b) If a set of vectors contains a scalar multiple of another vector, then the set is linearly dependent.

(c) If a set of vectors contains more vectors than there are entries in each vector, then the set is linearly dependent.

♡

The last part of the theorem is particularly useful, as it allows us to quickly determine if a set of vectors is linearly dependent by checking if one of the vectors is a linear combination of the others. We can formalise this idea in the following corollary.

Corollary 8.1

An indexed set $S = \{\boldsymbol{v}_1, \dots, \boldsymbol{v}_p\}$ of two or more vectors is linearly dependent if and only if at least one of the vectors in S is a linear combination of the others. If S is linearly dependent and $\boldsymbol{v}_1 \neq \mathbf{0}$, then some $\boldsymbol{v}_j$ (with $1 < j \leq p$) is a linear combination of the preceding vectors $\boldsymbol{v}_1, \dots, \boldsymbol{v}_{j-1}$.

♡

Remark: The corollary tells us that if a set of vectors is linearly dependent, then at least one of the vectors is redundant, as it can be expressed as a linear combination of the others. This redundancy is what causes the linear dependence. This does not mean that every vector in S is a linear combination of the others, but only that *at least one* vector in a linearly dependent set is a linear combination of others.

Example 8.17 Let

$$\boldsymbol{u} = \begin{bmatrix} 2 \\ 1 \\ 1 \end{bmatrix}, \quad \boldsymbol{v} = \begin{bmatrix} 4 \\ 3 \\ 5 \end{bmatrix}, \quad \boldsymbol{w} = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix}, \quad \boldsymbol{z} = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}.$$

(a) Is any pair of these vectors (e.g. $\{\boldsymbol{u}, \boldsymbol{v}\}$) linearly dependent? Explain.

(b) Does the answer to part (a) tell us that $\{\boldsymbol{u}, \boldsymbol{v}, \boldsymbol{w}, \boldsymbol{z}\}$ is linearly independent?

(c) Is $\{\boldsymbol{u}, \boldsymbol{v}, \boldsymbol{w}, \boldsymbol{z}\}$ linearly dependent? You should be able to answer this question without any computation.

Solution:

(a) To determine if any pair of these vectors is linearly dependent, we check whether one vector is a scalar multiple of the other.

Checking the pair $\{\boldsymbol{u}, \boldsymbol{v}\}$:

Assume there exists a scalar k such that:

$$\boldsymbol{v} = k\boldsymbol{u}.$$

Compute k using the first component:

$$k = \frac{v_1}{u_1} = \frac{4}{2} = 2.$$

Verify with the second component:

$$v_2 = ku_2 \implies 3 = 2 \times 1 \implies 3 = 2.$$

Since $3 \neq 2$, $\boldsymbol{v}$ is not a scalar multiple of $\boldsymbol{u}$.

Checking the pair $\{\boldsymbol{u}, \boldsymbol{w}\}$:

Assume there exists a scalar k such that:

$$\boldsymbol{u} = k\boldsymbol{w}.$$

Compute k using the first component:

$$k = \frac{u_1}{w_1} = \frac{2}{1} = 2.$$

Verify with the second component:

$$u_2 = kw_2 \implies 1 = 2 \times 1 \implies 1 = 2.$$

Since $1 \neq 2$, $\boldsymbol{u}$ is not a scalar multiple of $\boldsymbol{w}$.

Checking the pair $\{\boldsymbol{w}, \boldsymbol{z}\}$:

Since $z_1 = 0$ and $w_1 = 1$, assuming $\boldsymbol{w} = k\boldsymbol{z}$ leads to:

$$w_1 = kz_1 \implies 1 = k \times 0 \implies 1 = 0,$$

which is a contradiction.

Conclusion: No pair of these vectors is linearly dependent.

(b) No, the fact that no pair is linearly dependent does not imply that the entire set is linearly independent. Linear independence requires that the only solution to:

$$c_1\boldsymbol{u} + c_2\boldsymbol{v} + c_3\boldsymbol{w} + c_4\boldsymbol{z} = \mathbf{0}$$

is $c_1 = c_2 = c_3 = c_4 = 0$.

(c) Yes, the set is linearly dependent.

In $\mathbb{R}^3$, any set of more than three vectors must be linearly dependent because the maximum number of linearly independent vectors in $\mathbb{R}^3$ is three.

Conclusion:

(a) No pair of these vectors is linearly dependent.

(b) The answer to part (a) does not guarantee that the entire set is linearly independent.

(c) The set $\{\boldsymbol{u}, \boldsymbol{v}, \boldsymbol{w}, \boldsymbol{z}\}$ is linearly dependent without the need for further computation.

◀

Chapter 9 Matrix Algebra

In the previous chapters, we explored concepts like linear equations, echelon forms, row reductions, linear independence, vector equations, and matrix equations. Now, to take our problem-solving skills to the next level, we need to dive into matrix algebra. Matrix operations allow us to handle complex systems more efficiently, building on the foundations we've established. The tools and techniques in this chapter will streamline the way we work with multiple matrices and lead us toward deeper insights, including key ideas like the Invertible Matrix Theorem.

9.1 Matrix Arithmetic

Matrix arithmetic like adding, subtracting, scaling, and multiplying let you combine and manipulate matrices in simple ways. These operations are the building blocks for more complex matrix manipulations, like solving systems of equations and finding inverses.

Addition and Scalar Multiplication

We begin with the definition of matrix addition.

Definition 9.1 (Matrix Addition)

Let A and B be $m \times n$ matrices. The **sum** of A and B, denoted $A+B$, is the $m \times n$ matrix whose entries are obtained by adding the corresponding entries of A and B.
Given matrices

$$A = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \vdots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix}, \quad B = \begin{bmatrix} b_{11} & b_{12} & \cdots & b_{1n} \\ b_{21} & b_{22} & \cdots & b_{2n} \\ \vdots & \vdots & \vdots & \vdots \\ b_{m1} & b_{m2} & \cdots & b_{mn} \end{bmatrix}$$

both of the same dimension $m \times n$, the sum $A+B$ is thus defined as

$$A + B = \begin{bmatrix} a_{11}+b_{11} & a_{12}+b_{12} & \cdots & a_{1n}+b_{1n} \\ a_{21}+b_{21} & a_{22}+b_{22} & \cdots & a_{2n}+b_{2n} \\ \vdots & \vdots & \vdots & \vdots \\ a_{m1}+b_{m1} & a_{m2}+b_{m2} & \cdots & a_{mn}+b_{mn} \end{bmatrix}$$

♣

Remark: Similar considerations apply for substraction of matrices, although not mentioned here explicitly.

Next is the definition of scalar-matrix multiplication.

Definition 9.2 (Scalar-Matrix Multiplication)

Let A be an $m \times n$ matrix and c be a scalar. The **product** of c and A, denoted cA, is the $m \times n$ matrix whose entries are obtained by multiplying each entry of A by c, and is thus defined as

$$cA = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \vdots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix} = \begin{bmatrix} ca_{11} & ca_{12} & \cdots & ca_{1n} \\ ca_{21} & ca_{22} & \cdots & ca_{2n} \\ \vdots & \vdots & \vdots & \vdots \\ ca_{m1} & ca_{m2} & \cdots & ca_{mn} \end{bmatrix}$$

♣

Example 9.1 Given matrices

$$A = \begin{bmatrix} -5 & 2 & 0 \\ 7 & -3 & 4 \\ -1 & 3 & 2 \end{bmatrix}, \quad B = \begin{bmatrix} 0 & -1 & 8 \\ 6 & -14 & 2 \\ 9 & 5 & 1 \end{bmatrix}$$

find the sum $A + B$ and the product $3A$.

Solution: The sum $A + B$ is obtained by adding the corresponding entries of A and B:

$$\begin{aligned} A + B &= \begin{bmatrix} -5 & 2 & 0 \\ 7 & -3 & 4 \\ -1 & 3 & 2 \end{bmatrix} + \begin{bmatrix} 0 & -1 & 8 \\ 6 & -14 & 2 \\ 9 & 5 & 1 \end{bmatrix} \\ &= \begin{bmatrix} (-5)+(0) & (2)+(-1) & (0)+(8) \\ (7)+(6) & (-3)+(-14) & (4)+(2) \\ (-1)+(9) & (3)+(5) & (2)+(1) \end{bmatrix} \\ &= \begin{bmatrix} -5 & 1 & 8 \\ 13 & -17 & 6 \\ 8 & 8 & 3 \end{bmatrix} \end{aligned}$$

The product $3A$ is obtained by multiplying each entry of A by 3:

$$\begin{aligned} 3A &= 3\begin{bmatrix} -5 & 2 & 0 \\ 7 & -3 & 4 \\ -1 & 3 & 2 \end{bmatrix} \\ &= \begin{bmatrix} 3(-5) & 3(2) & 3(0) \\ 3(7) & 3(-3) & 3(4) \\ 3(-1) & 3(3) & 3(2) \end{bmatrix} \\ &= \begin{bmatrix} -15 & 6 & 0 \\ 21 & -9 & 12 \\ -3 & 9 & 6 \end{bmatrix} \end{aligned}$$

◀

Example 9.2 Given A and B below, find $3A - 2B$.

$$A = \begin{bmatrix} 1 & -2 & 5 \\ 0 & -3 & 9 \\ 4 & -6 & 7 \end{bmatrix}, B = \begin{bmatrix} 5 & 0 & -11 \\ 3 & -5 & 1 \\ -1 & -9 & 0 \end{bmatrix}$$

Solution: We compute:

$$3A - 2B = \begin{bmatrix} 3 & -6 & 15 \\ 0 & -9 & 27 \\ 12 & -18 & 21 \end{bmatrix} - \begin{bmatrix} 10 & 0 & -22 \\ 6 & -10 & 2 \\ -2 & -18 & 0 \end{bmatrix}$$

$$= \begin{bmatrix} -7 & -6 & 37 \\ -6 & 1 & 25 \\ 14 & 0 & 21 \end{bmatrix}$$

◀

Before moving on to matrix multiplication, we need to state some basic algebraic properties of matrix addition and scalar multiplication.

Theorem 9.1

Let A, B, C be matrices of the same size and let α, β be scalars. Then

(a) $A + B = B + A$

(b) $(A + B) + C = A + (B + C)$

(c) $A + 0 = A$

(d) $\alpha(A + B) = \alpha A + \alpha B$

(e) $(\alpha + \beta)A = \alpha A + \beta A$

(f) $\alpha(\beta A) = (\alpha\beta)A$

♡

Matrix Multiplication

Matrix multiplication is a bit more complex than addition and scalar multiplication. The product of two matrices is defined only when the number of columns in the first matrix is equal to the number of rows in the second matrix. The product of two matrices A and B is a new matrix C whose entries are determined by the dot product of the rows of A and the columns of B. Note that $\boldsymbol{x}$ is a column vector.

Let B be an $n \times p$ matrix and A be an $m \times n$ matrix. If $\boldsymbol{x} \in \mathbb{R}^p$, then multiplying B by $\boldsymbol{x}$ produces a new vector $B\boldsymbol{x}$ in $\mathbb{R}^n$. Once we have this result, we can further multiply it by A, giving us $A(B\boldsymbol{x})$, which is a vector in $\mathbb{R}^m$.

Thus, for any vector $\boldsymbol{x}$ in $\mathbb{R}^p$, this process produces a corresponding vector in $\mathbb{R}^m$. This two-step operation—first multiplying by B and then by A—is referred to as the composition of A and B, and is usually written as AB. Therefore, we have:

$$(AB)\boldsymbol{x} = A(B\boldsymbol{x})$$

To compute the matrix resulting from this composition, we multiply A by B directly, following the rules of matrix multiplication. The result is a matrix $C = AB$, where each entry of C is determined by the interactions between the rows of A and the columns of B.

In essence, C represents the matrix that captures the combined effect of applying both B and A to any vector $\boldsymbol{x} \in \mathbb{R}^p$, without needing to break it down into intermediate steps.

Definition 9.3

For $A \in \mathbb{R}^{m\times n}$ and $B \in \mathbb{R}^{n\times p}$, with $B = \begin{bmatrix} \boldsymbol{b_1} & \boldsymbol{b_2} \cdots & \boldsymbol{b_p} \end{bmatrix}$, we define the product AB by the formula

$$AB = \begin{bmatrix} A\boldsymbol{b_1} & A\boldsymbol{b_2} & \cdots & A\boldsymbol{b_p} \end{bmatrix}$$

♣

The product AB is defined only when the number of columns of A equals the number of rows of B. The following diagram is useful for remembering this:

$$\begin{tikzcd}[column sep=tiny] (m \arrow[rrrr, leftrightarrow, bend left, red] & \times n) \arrow[rr, leftrightarrow, bend right, blue] & \cdot & (n \times & p) & \rightarrow m \times p \end{tikzcd}$$

The diagram shows that the number of columns in the first matrix must equal the number of rows in the second matrix (the blue arrow). The result is a new matrix with the number of rows from the first matrix and the number of columns from the second matrix (the red arrow).

Example 9.3 For A and B below compute AB and BA.

$$A = \begin{bmatrix} 1 & 2 & -2 \\ 1 & 1 & -3 \end{bmatrix}, \quad B = \begin{bmatrix} -4 & 2 & 4 & -4 \\ -1 & -5 & -3 & 3 \\ -4 & -4 & -3 & -1 \end{bmatrix}$$

Solution: First $AB = \begin{bmatrix} A\boldsymbol{b_1} & A\boldsymbol{b_2} & A\boldsymbol{b_3} & A\boldsymbol{b_4} \end{bmatrix}$:

$$\begin{aligned}
AB &= \begin{bmatrix} 1 & 2 & -2 \\ 1 & 1 & -3 \end{bmatrix} \begin{bmatrix} -4 & 2 & 4 & -4 \\ -1 & -5 & -3 & 3 \\ -4 & -4 & -3 & -1 \end{bmatrix} \\
&= \left[\boxed{\begin{matrix} 2 \\ 7 \end{matrix}} \quad \begin{matrix} \text{Row 1 of } A \text{ and Column 1 of } B \\ \text{Row 2 of } A \text{ and Column 1 of } B \end{matrix} \right. \\
&= \left[\begin{matrix} 2 \\ 7 \end{matrix} \boxed{\begin{matrix} 0 \\ 9 \end{matrix}} \quad \begin{matrix} \text{Row 1 of } A \text{ and Column 2 of } B \\ \text{Row 2 of } A \text{ and Column 2 of } B \end{matrix} \right. \\
&= \left[\begin{matrix} 2 & 0 \\ 7 & 9 \end{matrix} \boxed{\begin{matrix} 4 \\ 10 \end{matrix}} \quad \begin{matrix} \text{Row 1 of } A \text{ and Column 3 of } B \\ \text{Row 2 of } A \text{ and Column 3 of } B \end{matrix} \right. \\
&= \left[\begin{matrix} 2 & 0 & 4 \\ 7 & 9 & 10 \end{matrix} \boxed{\begin{matrix} 4 \\ 2 \end{matrix}} \quad \begin{matrix} \text{Row 1 of } A \text{ and Column 4 of } B \\ \text{Row 2 of } A \text{ and Column 4 of } B \end{matrix} \right. \\
&= \begin{bmatrix} 2 & 0 & 4 & 4 \\ 7 & 9 & 10 & 2 \end{bmatrix}
\end{aligned}$$

On the other hand, BA is not defined! B has 4 columns and A has 2 rows. Thus, the number of columns in B is not equal to the number of rows in A. ◀

Example 9.4 Example Compute the matrix AB, where

$$A = \begin{bmatrix} 1 & 2 \\ 2 & 3 \\ 3 & 4 \end{bmatrix}, \quad \text{and } B = \begin{bmatrix} 2 & -1 \\ 3 & 1 \end{bmatrix}$$

Solution: By the definition of multiplication given above,

$$\begin{aligned}
AB &= \left[A\begin{bmatrix} 2 \\ 3 \end{bmatrix} A\begin{bmatrix} -1 \\ 1 \end{bmatrix} \right] \\
&= \left[\begin{bmatrix} 1 & 2 \\ 2 & 3 \\ 3 & 4 \end{bmatrix} \begin{bmatrix} 2 \\ 3 \end{bmatrix} \begin{bmatrix} 1 & 2 \\ 2 & 3 \\ 3 & 4 \end{bmatrix} \begin{bmatrix} -1 \\ 1 \end{bmatrix} \right] \\
&= \left[\begin{bmatrix} 1\cdot 2 + 2\cdot 3 \\ 2\cdot 2 + 3\cdot 3 \\ 3\cdot 2 + 4\cdot 3 \end{bmatrix} \begin{bmatrix} 1\cdot(-1) + 2\cdot 1 \\ 2\cdot(-1) + 3\cdot 1 \\ 3\cdot(-1) + 4\cdot 1 \end{bmatrix} \right] \\
&= \begin{bmatrix} 8 & 1 \\ 13 & 1 \\ 18 & 1 \end{bmatrix}
\end{aligned}$$

◀

Example 9.5 If A is 3×5 and B is 5×2, what are the sizes of AB and BA (assuming they are defined)?

Solution:

- The product AB is defined since the number of columns of A matches the number of rows of B (5). The resulting matrix AB is a 3×2 matrix.
- The product BA is not defined, since the number of columns of B (2) does not match the number of rows of A (3).

◀

The next example illustrate that even if both AB and BA are defined, they are not necessarily equal.

Example 9.6 For $A = \begin{bmatrix} -4 & 4 & 3 \\ 3 & -3 & -1 \\ -2 & -1 & 1 \end{bmatrix}$ and $B = \begin{bmatrix} -1 & -1 & 0 \\ -3 & 0 & -2 \\ -2 & 1 & -2 \end{bmatrix}$ compute AB and BA.

Solution: First AB:

$$\begin{aligned}
AB &= \begin{bmatrix} -4 & 4 & 3 \\ 3 & -3 & -1 \\ -2 & -1 & 1 \end{bmatrix} \begin{bmatrix} -1 & -1 & 0 \\ -3 & 0 & -2 \\ -2 & 1 & -2 \end{bmatrix} \\
&= \begin{bmatrix} -14 \\ 8 \\ 3 \end{bmatrix} \\
&= \begin{bmatrix} -14 & 7 \\ 8 & -4 \\ 3 & 3 \end{bmatrix} \\
&= \begin{bmatrix} -14 & 7 & -14 \\ 8 & -4 & 8 \\ 3 & 3 & 0 \end{bmatrix}
\end{aligned}$$

Next BA:

$$\begin{aligned}
BA &= \begin{bmatrix} -1 & -1 & 0 \\ -3 & 0 & -2 \\ -2 & 1 & -2 \end{bmatrix} \begin{bmatrix} -4 & 4 & 3 \\ 3 & -3 & -1 \\ -2 & -1 & 1 \end{bmatrix} \\
&= \begin{bmatrix} 1 & & \\ 16 & & \\ 15 & & \end{bmatrix} \\
&= \begin{bmatrix} 1 & -1 & \\ 16 & -10 & \\ 15 & -9 & \end{bmatrix} \\
&= \begin{bmatrix} 1 & -1 & -2 \\ 16 & -10 & -11 \\ 15 & -9 & -9 \end{bmatrix}
\end{aligned}$$

We see that $AB \neq BA$. ◀

In regular arithmetic the multiplicative identity is 1. In matrix algebra, the multiplicative identity is the identity matrix, denoted by I. The identity matrix is a *square* matrix with 1s on the diagonal and 0s elsewhere. The size of the identity matrix is determined by the context, and is usually clear from the context. For example, I_2 is a 2×2 identity matrix, and I_3 is a 3×3 identity matrix and in general $I_n \in \mathbb{R}^{n \times n}$ is an $n \times n$ identity matrix:

$$I_n = \begin{bmatrix} 1 & 0 & 0 & \cdots & 0 \\ 0 & 1 & 0 & \cdots & 0 \\ \vdots & \vdots & \vdots & \cdots & \vdots \\ 0 & 0 & 0 & \cdots & 1 \end{bmatrix}$$

We can now state the following theorem.

Theorem 9.2

Let A, B, C be matrices, of appropriate dimensions, and let α be a scalar. Then

(a) $A(BC) = (AB)C$

(b) $A(B + C) = AB + AC$

(c) $(B + C)A = BA + CA$

(d) $\alpha(AB) = (\alpha A)B = A(\alpha B)$

(e) $I_n A = AI_n = A$

♡

We conclude this section by looking at the kth power of a matrix.

Definition 9.4

Let A be a square matrix, i.e. $A \in \mathbb{R}^{n\times n}$. The kth power of A, denoted A^k, is defined as the product of A with itself k times. That is,

$$A^k = \underbrace{AAA\cdot\cdots A}_{k \text{ times}}$$

where A appears k times on the right-hand side. ♣

Example 9.7 Compute A^3 if

$$A = \begin{bmatrix} -2 & 3 \\ 1 & 0 \end{bmatrix}$$

Solution:

Compute A^2 :

$$A^2 = \begin{bmatrix} -2 & 3 \\ 1 & 0 \end{bmatrix}\begin{bmatrix} -2 & 3 \\ 1 & 0 \end{bmatrix} = \begin{bmatrix} 7 & -6 \\ -2 & 3 \end{bmatrix}$$

And then A^3 :

$$\begin{aligned} A^3 = A^2A &= \begin{bmatrix} 7 & -6 \\ -2 & 3 \end{bmatrix}\begin{bmatrix} -2 & 3 \\ 1 & 0 \end{bmatrix} \\ &= \begin{bmatrix} -20 & 21 \\ 7 & -6 \end{bmatrix} \end{aligned}$$

We could also do:

$$A^3 = AA^2 = \begin{bmatrix} -2 & 3 \\ 1 & 0 \end{bmatrix}\begin{bmatrix} 7 & -6 \\ -2 & 3 \end{bmatrix} = \begin{bmatrix} -20 & 21 \\ 7 & -6 \end{bmatrix}$$

◀

9.2 Matrix Transpose

We begin with the definition of the transpose of a matrix.

Definition 9.5

Given a matrix $A \in \mathbb{R}^{m\times n}$, the transpose of A is the matrix A^T whose ith column is the ith row of A. ♣

If A is $m \times n$ then A^T is $n \times m$. For example, if

$$A = \begin{bmatrix} 0 & -1 & 8 & -7 & -4 \\ -4 & 6 & -10 & -9 & 6 \\ 9 & 5 & -2 & -3 & 5 \\ -8 & 8 & 4 & 7 & 7 \end{bmatrix}$$

then

$$A^T = \begin{bmatrix} 0 & -4 & 9 & -8 \\ -1 & 6 & 5 & 8 \\ 8 & -10 & -2 & 4 \\ -7 & -9 & -3 & 7 \\ -4 & 6 & 5 & 7 \end{bmatrix}$$

Example 9.8 Compute $(AB)^T$ and B^TA^T if

$$A = \begin{bmatrix} -2 & 1 & 0 \\ 3 & -1 & -3 \end{bmatrix}, \quad B = \begin{bmatrix} -2 & 1 & 2 \\ -1 & -2 & 0 \\ 0 & 0 & -1 \end{bmatrix}.$$

Solution: First, compute $(AB)^T$:

$$\begin{aligned} AB &= \begin{bmatrix} -2 & 1 & 0 \\ 3 & -1 & -3 \end{bmatrix} \begin{bmatrix} -2 & 1 & 2 \\ -1 & -2 & 0 \\ 0 & 0 & -1 \end{bmatrix} \\ &= \begin{bmatrix} 3 & -4 & -4 \\ -5 & 5 & 9 \end{bmatrix} \end{aligned}$$

and then $(AB)^T$:

$$\begin{aligned} (AB)^T &= \begin{bmatrix} 3 & -4 & -4 \\ -5 & 5 & 9 \end{bmatrix}^T \\ &= \begin{bmatrix} 3 & -5 \\ -4 & 5 \\ -4 & 9 \end{bmatrix} \end{aligned}$$

Next, compute B^TA^T:

$$\begin{aligned} B^TA^T &= \begin{bmatrix} -2 & -1 & 0 \\ 1 & -2 & 0 \\ 2 & 0 & -1 \end{bmatrix} \begin{bmatrix} -2 & 3 \\ 1 & -1 \\ 0 & -3 \end{bmatrix} \\ &= \begin{bmatrix} 3 & -5 \\ -4 & 5 \\ -4 & 9 \end{bmatrix} \end{aligned}$$

We see that $(AB)^T = B^TA^T$. ◀

The following theorem summarises the properties of the transpose of a matrix.

Theorem 9.3

Let A and B be matrices of appropriate dimensions and let α be a scalar. Then

(a) $(A^T)^T = A$

(b) $(A + B)^T = A^T + B^T$

(c) $(\alpha A)^T = \alpha A^T$

(d) $(AB)^T = B^TA^T$

♡

A consequence of property (4) is that

$$(A_1A_2\dots A_k)^T = A_k^T A_{k-1}^T \cdots A_2^T A_1^T$$

and as a special case

$$\left(A^k\right)^T = \left(A^T\right)^k$$

9.3 Invertible Matrices

The inverse of a square matrix $A \in \mathbb{R}^{n\times n}$ extends the concept of the reciprocal for a nonzero real number $a \in \mathbb{R}$. More precisely, the inverse of a non-zero number $a \in \mathbb{R}$ is the unique number $c \in \mathbb{R}$ such that $ac = ca = 1$. The inverse of $a \neq 0$, typically written as $a^{-1} = \frac{1}{a}$, enables solving the equation $ax = b$:

$$ax = b \Rightarrow a^{-1}ax = a^{-1}b \Rightarrow x = a^{-1}b.$$

This concept extends to square matrices, where the inverse of a matrix $A \in \mathbb{R}^{n\times n}$ is a matrix $C \in \mathbb{R}^{n\times n}$ such that $AC = CA = I_n$, where I_n is the identity matrix of size $n \times n$. The inverse of a matrix is denoted by A^{-1}. We can now define the invertible matrix.

Definition 9.6

A square matrix $A \in \mathbb{R}^{n\times n}$ is invertible (or **nonsingular**) if there exists a matrix $C \in \mathbb{R}^{n\times n}$ such that $AC = CA = I_n$. The matrix C is called the inverse of A and is denoted by A^{-1}. Thus, $A^{-1}A = AA^{-1} = I_n$. ♣

Example 9.9 Given A and C below, show that C is the inverse of A.

$$A = \begin{bmatrix} 1 & -3 & 0 \\ -1 & 2 & -2 \\ -2 & 6 & 1 \end{bmatrix}, \quad C = \begin{bmatrix} -14 & -3 & -6 \\ -5 & -1 & -2 \\ 2 & 0 & 1 \end{bmatrix}$$

Solution: We need to show that $AC = CA = I_3$. First, compute AC:

$$\begin{aligned} AC &= \begin{bmatrix} 1 & -3 & 0 \\ -1 & 2 & -2 \\ -2 & 6 & 1 \end{bmatrix} \begin{bmatrix} -14 & -3 & -6 \\ -5 & -1 & -2 \\ 2 & 0 & 1 \end{bmatrix} \\ &= \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} = I_3 \end{aligned}$$

Next, compute CA:

$$\begin{aligned} CA &= \begin{bmatrix} -14 & -3 & -6 \\ -5 & -1 & -2 \\ 2 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & -3 & 0 \\ -1 & 2 & -2 \\ -2 & 6 & 1 \end{bmatrix} \\ &= \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} = I_3 \end{aligned}$$

We see that $C = A^{-1}$ is the inverse of A. ◀

The following theorem summarizes the relationship between the matrix inverse and matrix multiplication and matrix transpose.

Theorem 9.4

Let A and B be invertible $n \times n$ matrices. Then:

(a) A^{-1} is invertible, with
$$\left(A^{-1}\right)^{-1} = A$$

(b) The product AB is invertible, with
$$(AB)^{-1} = B^{-1}A^{-1}$$

(c) The transpose of A is also invertible, i.e. A^T is invertible, with
$$\left(A^T\right)^{-1} = \left(A^{-1}\right)^T$$

♡

We will now consider how we can find the inverse of a matrix.

Finding the Determinant and Inverse of a 2×2 Matrix

The inverse of a matrix is not always easy to find. However, for a 2×2 matrix, we can use a formula to find the inverse.

Theorem 9.5

Let $A = \begin{bmatrix} a & b \\ c & d \end{bmatrix}$. If $ad - bc \neq 0$, then the inverse of A is given by
$$A^{-1} = \frac{1}{ad-bc}\begin{bmatrix} d & -b \\ -c & a \end{bmatrix}$$

♡

Remark: The quantity $ad - bc$ is called the determinant of A, and we write
$$\det(A) = |A| = ad - bc$$

The determinant of a 2×2 matrix is a scalar quantity that provides information about the matrix. If the determinant is zero, then the matrix is not invertible. If the determinant is non-zero, then the matrix is invertible. These formulae are only valid 2×2 matrices.

Example 9.10 We let
$$A = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$$
and compute A^{-1}. By the theorem, we have
$$\begin{aligned} A^{-1} &= \frac{1}{1\cdot 4 - 2\cdot 3}\begin{bmatrix} 4 & -2 \\ -3 & 1 \end{bmatrix} \\ &= -\frac{1}{2}\begin{bmatrix} 4 & -2 \\ -3 & 1 \end{bmatrix} \\ &= \begin{bmatrix} -2 & 1 \\ \frac{3}{2} & -\frac{1}{2} \end{bmatrix} \end{aligned}$$

Now we confirm that $AA^{-1} = I_2$:

$$
\begin{aligned}
AA^{-1} &= \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \begin{bmatrix} -2 & 1 \\ \frac{3}{2} & -\frac{1}{2} \end{bmatrix} \\
&= \begin{bmatrix} 1(-2) + 2\left(\frac{3}{2}\right) & 1(1) + 2\left(-\frac{1}{2}\right) \\ 3(-2) + 4\left(\frac{3}{2}\right) & 3(1) + 4\left(-\frac{1}{2}\right) \end{bmatrix} \\
&= \begin{bmatrix} -2+3 & 1-1 \\ -6+6 & 3-2 \end{bmatrix} \\
&= \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}
\end{aligned}
$$

Example 9.11 Find the inverse of $A = \begin{bmatrix} 1 & 3 \\ -1 & -2 \end{bmatrix}$ if it exists.

Solution: We first compute the determinant of A:

$$\det(A) = 1(-2) - 3(-1) = (-2) + 3 = 1$$

Since the determinant is non-zero, the inverse of A exists. By the formula, we have

$$A^{-1} = \frac{1}{1} \begin{bmatrix} -2 & -3 \\ 1 & 1 \end{bmatrix}$$

We can confirm that $AA^{-1} = I_2$:

$$AA^{-1} = \begin{bmatrix} 1 & 3 \\ -1 & -2 \end{bmatrix} \begin{bmatrix} -2 & -3 \\ 1 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} = I_2$$

◀

For larger matrices, we need to use other methods to find the inverse.

Finding the Inverse of a $n \times n$ Matrix

An **elementary matrix** is a matrix that is obtained by performing a single elementary row operation (replacement, swap, or scaling) on an identity matrix. For example, the matrix

$$E_1 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ -2 & 0 & 1 \end{bmatrix}$$

is an elementary matrix since it is obtained from I_3 via the single elementary row operation $r_3 \mapsto r_3 - 2r_1$.

Example 9.12

Let A be a general 3×3 matrix

$$A = \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix}$$

and we let E_1, E_2, and E_3 be the elementary matrices

$$E_1 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ -2 & 0 & 1 \end{bmatrix}, \quad E_2 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix}, \quad \text{and} \quad E_3 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 3 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Notice, then, that E_1 corresponds to $r_3 \mapsto r_3 - 2r_1, E_2$ corresponds to $r_2 \longleftrightarrow r_3$, and E_3 corresponds to $r_2 \mapsto 3r_2$. We also have the following products:

$$E_1A = \begin{bmatrix} a & b & c \\ d & e & f \\ g-2a & h-2b & i-2c \end{bmatrix}, \quad E_2A = \begin{bmatrix} a & b & c \\ g & h & i \\ d & e & f \end{bmatrix}, \quad \text{and} \quad E_3A = \begin{bmatrix} a & b & c \\ 3d & 3e & 3f \\ g & h & i \end{bmatrix}.$$

Notice that E_1A is the matrix obtained by performing the row operation $r_3 \mapsto r_3 - 2r_1$ on A. In general, multiplying A on the left by an elementary matrix is the same as performing the corresponding row operation on A. We can also represent a sequence of row operations by multiplication of several elementary matrices. For example,

$$E_2E_1A = \begin{bmatrix} a & b & c \\ g-2a & h-2b & i-2c \\ d & e & f \end{bmatrix}$$

corresponds to performing $r_3 \mapsto r_3 - 2r_1$ followed by $r_2 \longleftrightarrow r_3$ on the matrix A. As we have observed before, row operations are reversible. It follows that elementary matrices are also invertible. This leads to the following theorem.

Theorem 9.6

An $n \times n$ matrix A is invertible if and only if A is row equivalent to I_n. In this case, any sequence of elementary row operations that reduces A to I_n also transforms I_n into A^{-1}. ♡

This theorem leads to a nice algorithm for finding the inverse of an $n \times n$ matrix, assuming such an inverse exists.

Algorithm for Finding the Inverse of an $n \times n$ Matrix

Let A be an $n \times n$ matrix. To find the inverse of A, follow these steps:

1. Form the augmented matrix $[A|I_n]$.
2. Perform row operations on $[A|I_n]$ to reduce A to I_n.
3. The matrix on the right side of the augmented matrix is A^{-1}.

If A is not invertible, then the algorithm will not be able to reduce A to I_n.

Example 9.13 Find the inverse of $A = \begin{bmatrix} 1 & 0 & 3 \\ 1 & 1 & 0 \\ -2 & 0 & -7 \end{bmatrix}$ if it exists.

Solution: Solution. Form the augmented matrix $\left[\begin{array}{cc} A & I_3 \end{array}\right]$ and row reduce:

$$\left[\begin{array}{cccccc} 1 & 0 & 3 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 & 1 & 0 \\ -2 & 0 & -7 & 0 & 0 & 1 \end{array}\right] \quad \begin{array}{l} r_2 \mapsto -r_1 + r_2 \\ r_3 \mapsto 2r_1 + r_3 \end{array} \quad \left[\begin{array}{cccccc} 1 & 0 & 3 & 1 & 0 & 0 \\ 0 & 1 & -3 & -1 & 1 & 0 \\ 0 & 0 & -1 & 2 & 0 & 1 \end{array}\right]$$

$$r_3 \mapsto -r_3 \quad \left[\begin{array}{cccccc} 1 & 0 & 3 & 1 & 0 & 0 \\ 0 & 1 & -3 & -1 & 1 & 0 \\ 0 & 0 & 1 & -2 & 0 & -1 \end{array}\right]$$

$$\begin{array}{l} r_2 \mapsto 3r_3 + r_2 \\ r_1 \mapsto -3r_3 + r_1 \end{array} \quad \left[\begin{array}{cccccc} 1 & 0 & 0 & 7 & 0 & 3 \\ 0 & 1 & 0 & -7 & 1 & -3 \\ 0 & 0 & 1 & -2 & 0 & -1 \end{array}\right]$$

Therefore, $\operatorname{rref} A = I_3$, confirming that A is invertible. The inverse is

$$A^{-1} = \left[\begin{array}{ccc} 7 & 0 & 3 \\ -7 & 1 & -3 \\ -2 & 0 & -1 \end{array}\right]$$

Verify:

$$AA^{-1} = \left[\begin{array}{ccc} 1 & 0 & 3 \\ 1 & 1 & 0 \\ -2 & 0 & -7 \end{array}\right] \left[\begin{array}{ccc} 7 & 0 & 3 \\ -7 & 1 & -3 \\ -2 & 0 & -1 \end{array}\right] = \left[\begin{array}{ccc} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{array}\right]$$

◀

Example 9.14 Find the inverse of $A = \left[\begin{array}{ccc} 1 & 0 & 1 \\ 1 & 1 & -2 \\ -2 & 0 & -2 \end{array}\right]$ if it exists.

Solution: Form the augmented matrix $\left[\begin{array}{cc} A & I_3 \end{array}\right]$ and row reduce:

$$\left[\begin{array}{cccccc} 1 & 0 & 1 & 1 & 0 & 0 \\ 1 & 1 & -2 & 0 & 1 & 0 \\ -2 & 0 & -2 & 0 & 0 & 1 \end{array}\right] \quad \begin{array}{l} r_2 \mapsto -r_1 + r_2 \\ r_3 \mapsto 2r_1 + r_3 \end{array} \quad \left[\begin{array}{cccccc} 1 & 0 & 1 & 1 & 0 & 0 \\ 0 & 1 & -3 & -1 & 1 & 0 \\ 0 & 0 & 0 & 2 & 0 & 1 \end{array}\right]$$

We need not go further since the rref(A) is not I_3. Therefore, A is not invertible. ◀

Example 9.15 Find the inverse of the matrix

$$A = \left[\begin{array}{ccc} 1 & 2 & 1 \\ 4 & 5 & 3 \\ 0 & 0 & 2 \end{array}\right]$$

if it exists.

Solution: (We ommit the row reduction steps for brevity.)

$$\left[\begin{array}{cccccc} 1 & 2 & 1 & 1 & 0 & 0 \\ 4 & 5 & 3 & 0 & 1 & 0 \\ 0 & 0 & 2 & 0 & 0 & 1 \end{array}\right] \sim \left[\begin{array}{cccccc} 1 & 2 & 1 & 1 & 0 & 0 \\ 0 & -3 & -1 & -4 & 1 & 0 \\ 0 & 0 & 2 & 0 & 0 & 1 \end{array}\right]$$
$$\sim \left[\begin{array}{cccccc} 1 & 2 & 1 & 1 & 0 & 0 \\ 0 & -3 & -1 & -4 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & \frac{1}{2} \end{array}\right]$$
$$\sim \left[\begin{array}{cccccc} 1 & 2 & 0 & 1 & 0 & -\frac{1}{2} \\ 0 & -3 & 0 & -4 & 1 & \frac{1}{2} \\ 0 & 0 & 1 & 0 & 0 & \frac{1}{2} \end{array}\right]$$
$$\sim \left[\begin{array}{cccccc} 1 & 2 & 0 & 1 & 0 & -\frac{1}{2} \\ 0 & 1 & 0 & \frac{4}{3} & -\frac{1}{3} & -\frac{1}{6} \\ 0 & 0 & 1 & 0 & 0 & \frac{1}{2} \end{array}\right]$$
$$\sim \left[\begin{array}{cccccc} 1 & 0 & 0 & -\frac{5}{3} & \frac{2}{3} & -\frac{1}{6} \\ 0 & 1 & 0 & \frac{4}{3} & -\frac{1}{3} & -\frac{1}{6} \\ 0 & 0 & 1 & 0 & 0 & \frac{1}{2} \end{array}\right].$$

Hence

$$A^{-1} = \left[\begin{array}{ccc} -\frac{5}{3} & \frac{2}{3} & -\frac{1}{6} \\ \frac{4}{3} & -\frac{1}{3} & -\frac{1}{6} \\ 0 & 0 & \frac{1}{2} \end{array}\right].$$

◀

With the method for finding matrix inverses established, we now have a powerful tool for approaching linear systems. We will see how matrix inverses can be used to solve systems of equations efficiently, offering a straightforward way to find solutions when a matrix is invertible.

Solving Matrix Equations with Inverses

Just like we can solve an equation by using the reciprocal of a non-zero number as we saw previously, we can do something similar with matrices. When a matrix A is invertible, we can use its inverse to solve matrix equations involving A. This brings us to the theorem below, which shows how we can find solutions for equations like $A\boldsymbol{x} = \boldsymbol{b}$ when A has an inverse.

Theorem 9.7

Let A be an invertible matrix. Then the equation $A\boldsymbol{x} = \boldsymbol{b}$ has a unique solution given by $\boldsymbol{x} = A^{-1}\boldsymbol{b}$. ♡

Example 9.16 We can use the inverse from example 9.10 to solve the linear system

$$\begin{aligned} x_1 + 2x_2 &= 5 \\ 3x_1 + 4x_2 &= 6. \end{aligned}$$

We think of this in matrix terms as $A\boldsymbol{x} = \boldsymbol{b}$, where

$$A = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}, \quad \boldsymbol{x} = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}, \quad \text{and} \quad \boldsymbol{b} = \begin{bmatrix} 5 \\ 6 \end{bmatrix}$$

The theorem tells us that $\boldsymbol{x} = A^{-1}\boldsymbol{b}$ is a solution, i.e. that

$$\begin{aligned}\begin{bmatrix} x_1 \\ x_2 \end{bmatrix} &= \begin{bmatrix} -2 & 1 \\ \frac{3}{2} & -\frac{1}{2} \end{bmatrix}\begin{bmatrix} 5 \\ 6 \end{bmatrix} \\ &= \begin{bmatrix} -2(5) + 1(6) \\ \frac{3}{2}(5) - \frac{1}{2}(6) \end{bmatrix} \\ &= \begin{bmatrix} -4 \\ \frac{9}{2} \end{bmatrix}\end{aligned}$$

Example 9.17 Use the result from example 9.9 to solve the linear system $A\boldsymbol{x} = \boldsymbol{b}$ where

$$A = \begin{bmatrix} 1 & -3 & 0 \\ -1 & 2 & -2 \\ -2 & 6 & 1 \end{bmatrix}, \quad \boldsymbol{b} = \begin{bmatrix} 1 \\ -3 \\ -1 \end{bmatrix}$$

Solution: We showed in example 9.9 that

$$A^{-1} = \begin{bmatrix} -14 & -3 & -6 \\ -5 & -1 & -2 \\ 2 & 0 & 1 \end{bmatrix}$$

Therefore, the unique solution to the linear system $A\boldsymbol{x} = \boldsymbol{b}$ is

$$A^{-1}\boldsymbol{b} = \begin{bmatrix} -14 & -3 & -6 \\ -5 & -1 & -2 \\ 2 & 0 & 1 \end{bmatrix}\begin{bmatrix} 1 \\ -3 \\ -1 \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \\ 1 \end{bmatrix}$$

Verify:

$$\begin{bmatrix} 1 & -3 & 0 \\ -1 & 2 & -2 \\ -2 & 6 & 1 \end{bmatrix}\begin{bmatrix} 1 \\ 0 \\ 1 \end{bmatrix} = \begin{bmatrix} 1 \\ -3 \\ -1 \end{bmatrix}$$

◀

We conclude this chapter with one of the most important theorems in linear algebra, The Invertible Matrix Theorem.

9.4 The Invertible Matrix Theorem

The Invertible Matrix Theorem provides a comprehensive list of equivalent statements for a square matrix to be invertible and it enables us to determine the truth value of one statement by checking the truth value of another.

Theorem 9.8

Let A be an $n \times n$ matrix. The following statements are equivalent:

(a) A is invertible.

(b) A is row equivalent to I_n.

(c) A has n pivot positions (i.e. one for each row and column).

(d) The equation $A\boldsymbol{x} = \overline{0}$ has only the trivial solution.

(e) The columns of A are linearly independent.

(f) The equation $A\boldsymbol{x} = \boldsymbol{b}$ has a unique solution for each $\boldsymbol{b} \in \mathbb{R}^n$.

(g) The columns of $A \operatorname{span} \mathbb{R}^n$.

(h) $\det(A) \neq 0$.

(i) There is an $n \times n$ matrix C such that $CA = I$.

(j) There is an $n \times n$ matrix D such that $AD = I$.

(k) A^T is invertible.

♡

Note how the theorem connects the properties of a matrix to its invertibility, thus summarising the main elements of our discussion on linear systems, independence, vectors, and matrices.

Example 9.18 Decide if the following matrix is invertible:

$$A = \begin{bmatrix} 2 & 2 & 2 \\ 1 & 3 & 1 \\ 4 & 4 & 6 \end{bmatrix}$$

Solution: Performing row operations, we see

$$\begin{aligned} A &\sim \begin{bmatrix} 1 & 3 & 1 \\ 2 & 2 & 2 \\ 4 & 4 & 6 \end{bmatrix} \\ &\sim \begin{bmatrix} 1 & 3 & 1 \\ 0 & -4 & 0 \\ 0 & -8 & 2 \end{bmatrix} \\ &\sim \begin{bmatrix} 1 & 3 & 1 \\ 0 & -4 & 0 \\ 0 & 0 & 2 \end{bmatrix} \end{aligned}$$

which has 3 pivots, so by (c) we have that A is invertible. ◀

This concludes our discussion on matrices.

Chapter 10 Time Complexity

When discussing the efficiency of an algorithm, we aim to understand how it scales with input size, abstracting away details of specific hardware or programming environments. *Theoretical* time complexity, commonly represented by Big-$\mathcal{O}$ notation (with a capital $\mathcal{O}$, not a zero), enables this abstraction by focusing on the dominant terms of an algorithm's behaviour as inputs grow large. This notation, also known as Landau's symbol, was introduced by the German number theorist Edmund Landau, with $\mathcal{O}$, standing for "order," capturing the growth rate of a function.

This chapter will introduce the tools needed to analyse algorithmic efficiency using Big-$\mathcal{O}$, Omega (Ω), and Theta (Θ) notation.[1]

10.1 Introduction to Complexity Analysis

For instance, analysing an algorithm might reveal that the time it takes to complete a problem of size n is given by $T(n) = 4n^2 - 2n + 2$. By ignoring constants — since these vary with hardware and slower-growing terms - we simplify this to $T(n) = O\left(n^2\right)$, emphasizing the dominant term. This abstraction helps us understand complexity on large inputs, regardless of specific environments, making Big-$\mathcal{O}$ a powerful tool for comparing algorithms.

The two key components of this theoretical abstraction are:

1. **Ignoring multiplicative constants:** Constants often depend on factors like processor speed or compiler optimisations, making them impractical for broad analysis.
2. **Focusing on asymptotic behaviour:** We concentrate on the algorithm's behaviour with large inputs, as small cases typically run fast on any modern machine.

However, in *practical* time complexity, every term — including constants — plays a role. When running code on specific hardware with a known environment, those "negligible" constants in Big-$\mathcal{O}$ can become significant, especially for algorithms applied to small- and medium-sized data sets. So, while Big-$\mathcal{O}$ is crucial for high-level analysis and comparing algorithms, real-world complexity often requires looking at every term in the complexity expression. The following example illustrates this concept.

Algorithm 10.1: Linear Search in Python

```python
def linear_search(arr, target):
    for i in range(len(arr)):
        if arr[i] == target:
            return i
    return -1
```

[1]In practice, Big-$\mathcal{O}$ is often used informally as an umbrella term when discussing asymptotic analysis in a general sense, but it does not technically include Θ or Ω. Each notation has its own specific use case in bounding the function's growth rate from different perspectives (upper, lower, and tight bounds), as we will see shortly.

Algorithm 10.2: Linear Search in Java

```java
public static int linearSearch(int[] arr, int target) {
    for (int i = 0; i < arr.length; i++) {
        if (arr[i] == target) {
            return i;
        }
    }
    return -1;
}
```

This code performs a linear search through a list, checking each element until it finds the target or reaches the end of the list. Here, the time complexity is $\mathcal{O}(n)$ in the worst case, as every element must be checked. For small lists, the difference may be negligible, but as the input size grows, the linear relationship between input size and runtime becomes evident.

When we ask "How efficient is an algorithm or piece of code?", we note that efficiency encompasses various resources, including:

- CPU (time) usage
- Memory usage
- Disk usage
- Network usage

While all these resources are important, this chapter will primarily focus on time complexity, which concerns CPU usage.

1. **Performance:** This refers to how much time, memory, disk space, or other resources are actually used when a program is run. Performance is influenced by factors such as the machine, compiler, runtime environment, and other implementation details in addition to the code itself.
2. **Complexity:** Complexity, on the other hand, describes how the resource requirements of a program or algorithm scale with the size of the problem being solved. Complexity analysis focuses on the theoretical aspects of algorithm performance, characterizing how resource usage grows as input size increases.

Complexity impacts performance, but performance alone does not provide a complete picture of an algorithm's efficiency in terms of scalability. Understanding both is crucial to evaluating how well an algorithm will perform in diverse realworld scenarios, especially as input sizes grow significantly.

10.2 Growth of Functions

In Section 10.1the introduction to this chapter we noted that Big-$\mathcal{O}$ notation focuses on the dominant term of an algorithm's time complexity, abstracting away constants and smaller terms. This simplification is based on the idea that, as input sizes grow large, the dominant term will dictate the algorithm's complexity. Consider the following Java code snippet:

Algorithm 10.3: Java SumCalculator

```java
public static int f(int n) {
    int sum = 0;
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            sum += (i + j) / 6;
        }
    }
    return sum;
}
```

The code above is an implementation of the following expression:

$$\text{sum} = \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} \frac{i+j}{6}$$

Assuming n is a non-negative integer, let $T(n)$ represent the CPU time taken by a typical laptop to execute the function[2] $f(n)$, with input n. Naturally, $T(n)$ can vary due to factors that we may not fully control, such as the specific Java compiler, the hardware specifications of the laptop, the operating system, and other applications running simultaneously. However, we can be certain of the structure of the loops: the outer loop runs n times, and for each outer iteration, the inner loop also runs n times, resulting in a total of n^2 iterations.

Each iteration requires an approximate constant number of clock cycles, denoted c, which can differ depending on the system. Instead of stating that "$T(n)$ is approximately cn^2 nanoseconds, for some constant c that varies across systems," we use Big-$\mathcal{O}$ notation to write $T(n) = \mathcal{O}(n^2)$. This notation captures the idea that the CPU time grows at most quadratically with n. To visualize this, consider a graph of $T(n)$, which would generally appear parabolic with an approximate form of cn^2, where $c > 0$ is constant.

Big-$\mathcal{O}$ notation enables us to disregard constants, which often vary and lie beyond our control, yet do not affect the "family" or general growth rate classification of the function. For example, any potential $T(n)$ function in this case falls into the family of quadratic functions as illustrated in Figure 10.1. Additionally, lower-order terms become insignificant for large n as they can be bounded by a constant multiple of the dominant term.

For instance, if $f(n) = 3.5n^2 + 4n + 36$, we observe that $f(n) \geq 3.5n^2$. Simultaneously,

$$f(n) = 3.5n^2 + 4n + 36 \leq 3.5n^2 + 4n^2 + 36n^2 = 43.5n^2,$$

which effectively "sandwiches" $f(n)$ between two quadratic functions, confirming that its growth is essentially quadratic.

[2] In Java, it's technically a method, not a function. In Java terminology, a function that belongs to a class (the function would probably belong to a class such as `public class SumCalculator`) is called a method. A function generally refers to a standalone block of code that can be invoked independently, which is typical in languages like C or Python.

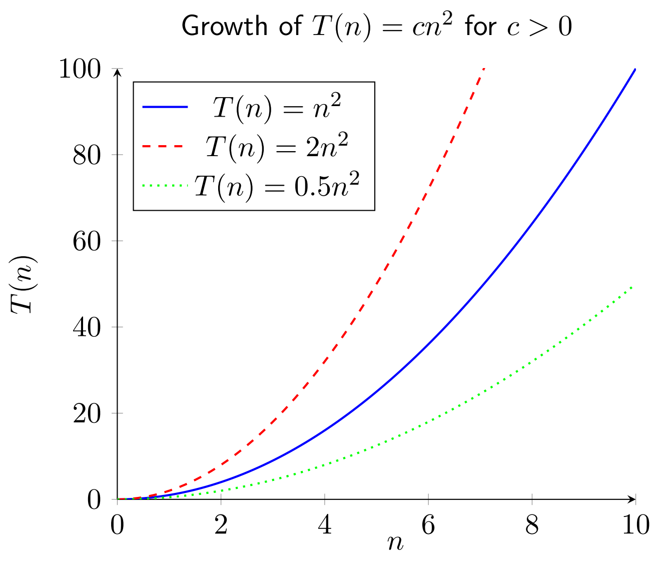


Figure 10.1: Graph of $T(n) = cn^2$ illustrating quadratic growth for different values of c

Primitive Functions

To understand how functions grow, let's arrange some fundamental functions by their growth rates. We are looking for relationships of the form $f(n) \leq g(n)$, but since we only care about behaviour with large inputs, these relationships hold only beyond some lower bound k for n.

Starting with the basics, any constant function grows more slowly than a linear function, simply because a constant does not grow at all. For large enough values, a linear function grows slower than a quadratic, while a cubic function grows faster than a quadratic.

Exponentials are well known for their rapid growth. Consider the tale of a company that decided to reward an employee by doubling their bonus each day for a month. Although it started with a modest amount, the exponential increase quickly escalated, and by the end of the month, the bonus would have been astronomical. Realizing the unsustainable cost, the company swiftly revised the agreement to a fixed rate instead. Mathematically, this effect is clear: $n^2 < 2^n$ for any integer $n \geq 4$. More generally, for any exponent k, $n^k < 2^n$ for sufficiently large n.

Even more dramatically, factorial growth, represented by $n!$, outpaces exponential growth for large n. The reasoning is that 2^n consists of n factors of 2, while $n!$ includes the first n integers as factors, most of which are larger than 2. Thus, $n!$ eventually grows faster than 2^n.

To summarize, exponential growth is faster than any polynomial growth, and factorial growth surpasses both. Using 1 as our constant function, we can express these relationships as:

$$1 \prec n \prec n^2 \prec n^3 \ldots \prec 2^n \prec n!$$

The symbol $\prec$ is used here because this ordering is not a standard algebraic inequality and only applies when n is sufficiently large.

For designing computer programs, only the first few of these growth rates are practical. Third-order polynomials can already grow too quickly for larger inputs, and exponential-time algorithms are only feasible for very small cases. Often, exponential algorithms are replaced by faster, approximate methods such as statistical sampling.

Now, let's consider functions with slower growth — these are the types of functions we aim for in efficient algorithms. Algorithms for searching in large datasets often have running times proportional to $\log n$. If we plot $\log n$ for values greater than 1, it shows slow but consistent growth, much slower than n itself.

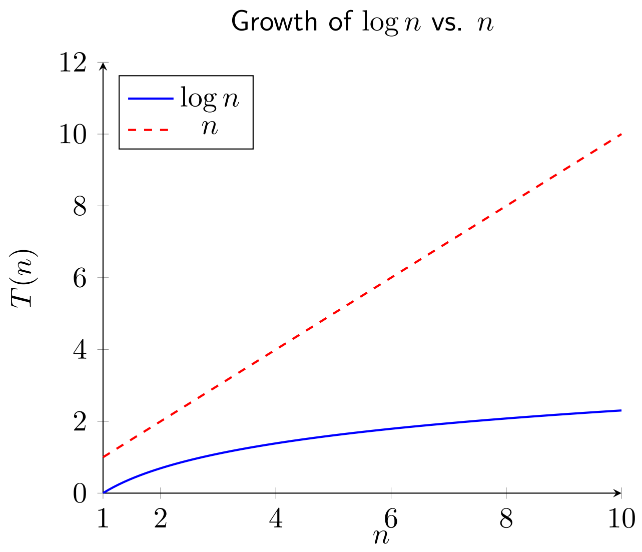


Figure 10.2: Graph comparing the growth rates of $\log n$ and n

Sorting algorithms, on the other hand, often have running times proportional to $n \log n$. For large n, we observe that $1 < \log n < n$, leading to $n < n \log n < n^2$. This order can be summarized as:

$$1 \prec \log n \prec n \prec n \log n \prec n^2$$

Remark: When discussing $\log n$, we typically refer to the base-2 logarithm, as it is the most common in computer science and software engineering. However, the base of the logarithm does not affect the growth rate, as it only introduces a constant factor. For example, $\log_2 n$ and $\log_{10} n$ grow at the same rate, differing only by a constant multiple.

Complexity	Growth Rate	Example Algorithms
$\mathcal{O}(1)$	Constant	Accessing an array element
$\mathcal{O}(\log n)$	Logarithmic	Binary search
$\mathcal{O}(n)$	Linear	Linear search
$\mathcal{O}(n \log n)$	Linearithmic (or Log-Linear)	MergeSort, QuickSort, HeapSort
$\mathcal{O}(n^2)$	Quadratic	Bubble Sort, Selection Sort
$\mathcal{O}(n^3)$	Cubic	Matrix multiplication
$\mathcal{O}(n^k)$	Polynomial	Polynomial-time approximation algorithms
$\mathcal{O}(c^n)$	Exponential	Subset sum, brute force for combinatorial problems
$\mathcal{O}(n!)$	Factorial	Brute force for travelling salesperson
$\mathcal{O}(n^n)$	Super-exponential	Recursive algorithms with high branching factor

Table 10.1: Growth rates of common functions and example algorithms

Table 10.1 summarizes the growth rates of common functions and supply some examples of algorithms that exhibit these growth rates and Figure 10.3 provides a visual representation of these growth rates. Memorizing the relative order of these common functions is valuable, as they will reappear frequently in this and other software engineering contexts.

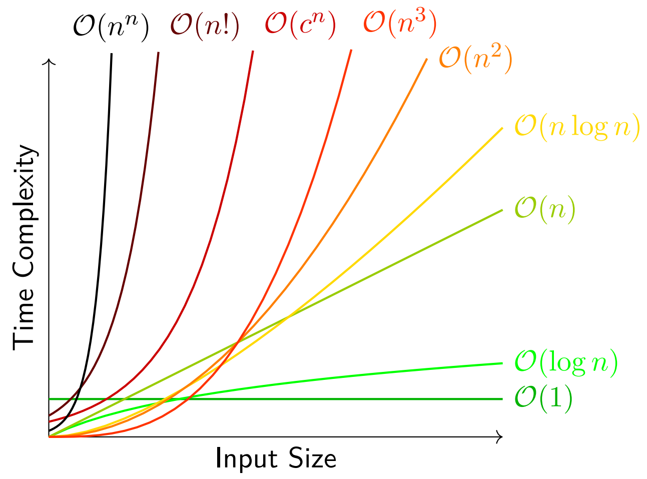


Figure 10.3: Conceptual illustration of time complexities

In the next section, we introduce the framework for analysing algorithms using Big-$\mathcal{O}$, Omega (Ω), and Theta (Θ) notation, which allows us to describe the growth behaviour of functions more precisely. These notations correspond to the worst-case, best-case, and tight bounds of an algorithm's time complexity, respectively. Note that Theta (Θ) notation is used when the upper and lower bounds of an algorithm's runtime are asymptotically the same, providing a precise description of the algorithm's complexity across different input sizes.

10.3 Asymptotic Analysis

In the analysis of algorithms, Big-$\mathcal{O}$, Omega (Ω), and Theta (Θ) notation each offer a unique perspective on an algorithm's efficiency, helping us understand how it will perform under different conditions. Whether we aim to determine the upper bound (worst-case scenario), lower bound (best-case scenario), or a tight bound, we can generally follow a systematic approach as follows:

1. **Identify the Dominant Operations:** Focus on the operations that contribute most significantly to the running time, particularly those that grow the fastest as the input size increases.
2. **Discard Constants and Lower-Order Terms:** For Big-$\mathcal{O}$ and Θ notation, ignore constant factors and lower-order terms to concentrate on the term that dictates the overall growth rate. For Ω, ensure that you establish a lower bound that is still representative of the growth rate.
3. **Express Complexity in Terms of the Input Size:** Write the complexity using n, the input size. If the runtime is proportional to the square of the input size, express it as $\mathcal{O}(n^2)$ if analysing an upper bound, $\Omega(n^2)$ if analysing a lower bound, or $\Theta(n^2)$ if establishing a tight bound.

Example 10.1

Consider the function, $f(n) = 4n^3 + 3n^2 + 2n + 10$. Let us determine the worst-case scenario, Big-$\mathcal{O}$, using the systematic approach outlined.

1. **Identify the Dominant Operations:**
   - Look at each term in $f(n) = 4n^3 + 3n^2 + 2n + 10$. The terms $4n^3, 3n^2, 2n$, and 10 contribute differently to the running time. As n grows, the $4n^3$ term will dominate the growth of the function because n^3 grows faster than n^2, n, and constant terms. Therefore, we focus on $4n^3$.
2. **Discard Constants:**
   - Ignore the constant coefficient 4 in $4n^3$, as Big-$\mathcal{O}$ notation only considers the asymptotic growth rate. We can disregard the lower-order terms $3n^2, 2n$, and 10 since they do not impact the asymptotic behaviour as n becomes very large.
3. **Express Complexity in Terms of the Input Size n:**
   - With the constants and lower-order terms removed, we are left with n^3. Since $4n^3$ dominates the growth, we describe the complexity in terms of n^3 for all three notations
     $$f(n) = \mathcal{O}\left(n^3\right)$$

Similar considerations apply to the best-case scenario, Omega (Ω), and the tight bound, Theta (Θ). We now introduce the three notations more in depth, starting with the tight bound notation, Theta (Θ).

Theta (Θ) Notation

Theta notation provides a *tight bound*, indicating that the growth rate of a function is bounded both above and below by the same asymptotic expression. In other words, Θ notation is used when the upper and lower bounds of an algorithm's runtime are the same, describing the algorithm's complexity consistently across different input sizes.

Mathematically, $T(n) = \Theta(f(n))$ if there exist constants $c_1, c_2 > 0$ and $k \geq 0$ such that $c_1 \cdot f(n) \leq T(n) \leq c_2 \cdot f(n)$ for all $n \geq k$.

We can summarize this in the following definition:

Definition 10.1 (Theta(Θ))

$f(n)$ is $\Theta(g(n))$ if there exist positive constants c_1, c_2, and k such that for all $n \geq k$,

$$c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$$

This means that $f(n)$ is asymptotically bounded both above and below by $g(n)$, meaning that $f(n)$ grows at the same rate as $g(n)$ as n becomes sufficiently large.

♣

Note that by writing "n becomes sufficiently large" we mean that the above inequalities do not have to hold for all $n \geq 0$, but rather for all $n \geq k$, for some constant $k \geq 0$. Sometimes we provide an exact value for k, and other times use the "sufficiently large" phrase when the value of k is not important.

Figure 10.4 illustrates the concept of Θ notation, showing how $f(n)$ is bounded both above and below by $c_1 g(n)$ and $c_2 g(n)$, respectively. The value of k shown is the minimum possible value; any greater value would also work. Theta provides both an upper and lower bound for the running time of an algorithm and describes the asymptotic behaviour of the algorithm's runtime, indicating that for sufficiently large inputs, the growth rate will stay within these bounds regardless of best-case or worst-case complexity. Or stated more succinctly:

$$f(n) = \Theta(g(n)): \quad f \text{ grows at the same rate as } g$$

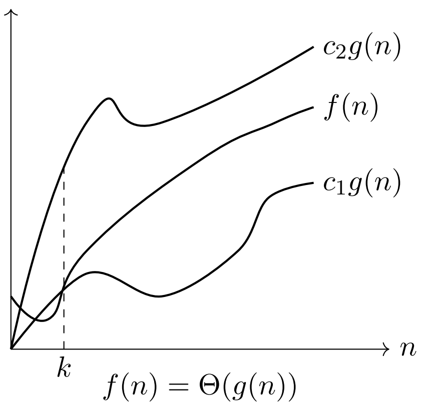


Figure 10.4: Illustration of Θ-notation with bounds on $f(n)$ by $c_1g(n)$ and $c_2g(n)$

Example 10.2

From the above discussion we have $f(n) = 3.5n^2 + 4n + 36 = \Theta\left(n^2\right)$ since

$$3.5n^2 \le f(n) = 3.5n^2 + 4n + 36 \le 3.5n^2 + 4n^2 + 36n^2 = 43.5n^2,$$

is true for all $n \ge 1$. And so $f(n) = \Theta\left(n^2\right)$ by Definition 10.3 , where $c_1 = 3.5$ and $c_2 = 43.5$.

The following properties hold for Θ notation:

- If $f(n) = \Theta(g(n))$, then $g(n) = \Theta(f(n))$.
- If $f(n) = \Theta(g(n))$ and $g(n) = \Theta(h(n))$, then $f(n) = \Theta(h(n))$.

Also, the following proposition provides a convenient way to show that two functions have the same order of growth.

Proposition 10.1

Let $f(n)$ and $g(n)$ be two nonnegative functions, and suppose there is a constant $c > 0$ for which

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = c$$

Then $f(n) = \Theta(g(n))$. Also, if

$$\lim_{n\to\infty} \frac{f(n)}{g(n)}$$

exists but does **not** equal a positive constant, then $f(n) \ne \Theta(g(n))$.

♠

Example 10.3 Use Proposition 10.1 to prove that if $f(n) = 3n^2 + 6n + 7$, then $f(n) = \Theta\left(n^2\right)$. In other words, $f(n)$ has quadratic growth.

Solution: Let $g(n) = n^2$. According to Proposition 10.1, we need to compute the limit:

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{3n^2 + 6n + 7}{n^2}.$$

Dividing each term in the numerator by n^2, we get:

$$\lim_{n\to\infty} \frac{3n^2}{n^2} + \frac{6n}{n^2} + \frac{7}{n^2} = \lim_{n\to\infty} \left(3 + \frac{6}{n} + \frac{7}{n^2}\right).$$

Now, as $n \to \infty$, the terms $\frac{6}{n}$ and $\frac{7}{n^2}$ approach 0, so we have:

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 3.$$

Since this limit is a positive constant ($c = 3 > 0$), by Proposition 10.1, it follows that $f(n) = \Theta(n^2)$.

Thus, we have shown that $f(n) = 3n^2 + 6n + 7$ has quadratic growth, as required. ◀

Upper and Lower Bound

Suppose we define a function $f(n)$ as follows:

$$f(n) = \begin{cases} 2n & \text{if } n \text{ is even} \\ 3n^2 & \text{if } n \text{ is odd} \end{cases}$$

What is the growth rate of $f(n)$? Unfortunately, we can't simply categorize $f(n)$ as having either linear or quadratic growth. This is because the limits

$$\lim_{n\to\infty} \frac{f(n)}{n}$$

and

$$\lim_{n\to\infty} \frac{f(n)}{n^2}$$

do not exist, as $f(n)$ continuously alternates between linear and quadratic behavior, switching from quadratic growth to linear growth and back. Therefore, the best we can do is to establish upper and lower bounds for $f(n)$.

For Big-$\mathcal{O}$: $f(n) = \mathcal{O}(g(n))$ if $f(n)$ does not grow faster than $g(n)$. In other words, $f(n) \leq c \cdot g(n)$ for some constant $c > 0$ and for all $n \geq k$, where $k \geq 0$ is a constant.

For Big-Omega: $f(n) = \Omega(g(n))$ if $f(n)$ grows at least as quickly as $g(n)$. That is, $f(n) \geq c \cdot g(n)$ for some constant $c > 0$ and for all $n \geq k$, where $k \geq 0$ is a constant.

Using these definitions, we find that $f(n) = \mathcal{O}(n^2)$ and $f(n) = \Omega(n)$. We formalise these considerations in the following sections.

Big-$\mathcal{O}$ Notation

Big-$\mathcal{O}$ notation provides an *upper bound* on the growth rate of a function. It describes the worst-case scenario for an algorithm's time complexity, indicating how the runtime or space requirements increase as the input size, n, grows. Big-$\mathcal{O}$ is particularly useful when comparing algorithms, as it allows us to focus on the most significant term that dictates complexity with large inputs.

Definition 10.2 (Big-$\mathcal{O}$)

$f(n)$ is $\mathcal{O}(g(n))$ if there exist positive constants c and k such that for all $n \geq k$,

$$f(n) \leq c \cdot g(n)$$

This means that $f(n)$ is asymptotically bounded above by $g(n)$, meaning that $f(n)$ grows at most as fast as $g(n)$ as n becomes sufficiently large. ♣

The expression "n becomes sufficiently large" implies that the inequality does not have to hold for all $n \geq 0$, but rather for all $n \geq k$, where k is a constant. Often, we may not need the exact value of k, so we use "sufficiently large" when the exact value is unimportant.

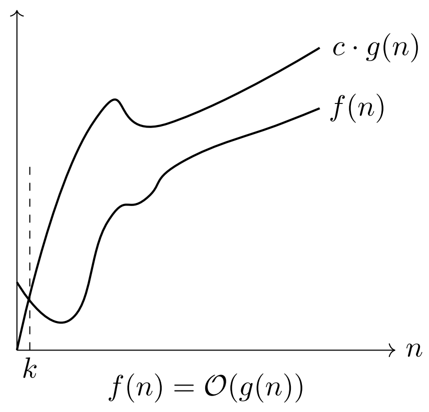


Figure 10.5: Illustration of $\mathcal{O}$-notation with $f(n)$ bounded above by $c \cdot g(n)$.

Figure 10.5 illustrates the concept of Big-$\mathcal{O}$ notation, showing that $f(n)$ is bounded above by $c \cdot g(n)$. This notation provides an upper bound for the running time of an algorithm, describing the asymptotic behaviour of the algorithm's runtime. It indicates that for sufficiently large inputs, the growth rate of $f(n)$ will not exceed this bound, regardless of the specific conditions or best-case complexity. Or stated more succinctly:

$$f(n) = \mathcal{O}(g(n)) : \quad f \text{ grows at most as fast as } g$$

Example 10.4 Using Definition 10.3, show that $f(n) = 3n^2 + 6n + 7$ is $\mathcal{O}(n^2)$. In other words, $f(n)$ has at most quadratic growth.

Solution: Let $g(n) = n^2$. We need to find constants $c > 0$ and k such that $f(n) \leq c \cdot g(n)$ for all $n \geq k$.

We can write:

$$f(n) = 3n^2 + 6n + 7 \leq 3n^2 + 6n^2 + 7n^2 = 16n^2.$$

Here, $c = 16$ and $k = 1$ work for the inequality, so we conclude $f(n) = \mathcal{O}(n^2)$. ◀

The following properties hold for $\mathcal{O}$ notation:

- If $f(n) = \mathcal{O}(g(n))$ and $g(n) = \mathcal{O}(h(n))$, then $f(n) = \mathcal{O}(h(n))$.
- Constant functions are $\mathcal{O}(1)$, meaning their growth is bounded regardless of input size.

Omega (Ω) Notation

Omega notation provides a *lower bound* on the growth rate, describing the best-case scenario of an algorithm's complexity. This notation is often used to indicate a guaranteed minimum time complexity, giving insight into how efficiently an algorithm can perform under ideal conditions.

Definition 10.3 (Omega Notation)

$f(n)$ is $\Omega(g(n))$ if there exist positive constants c and k such that for all $n \geq k$,

$$f(n) \geq c \cdot g(n)$$

This means that $f(n)$ is asymptotically bounded below by $g(n)$, meaning that $f(n)$ grows at least as fast as $g(n)$ as n becomes sufficiently large. ♣

The phrase "n becomes sufficiently large" indicates that the above inequality does not need to hold for all $n \geq 0$, but only for $n \geq k$ for some constant k.

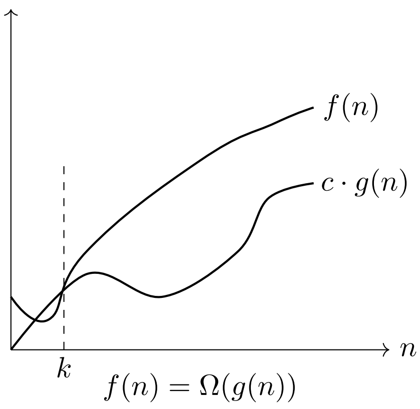


Figure 10.6: Illustration of Ω-notation showing $f(n)$ is asymptotically bounded below by $c \cdot g(n)$ for large values of n.

Figure 10.6 shows the concept of Omega notation, illustrating that $f(n)$ is bounded below by $c \cdot g(n)$. This notation provides a lower bound on the running time of an algorithm, describing the asymptotic growth rate of the algorithm's runtime in the best-case scenario. Or stated more succinctly:

$$f(n) = \Omega(g(n)) : \quad f \text{ grows at least as fast as } g$$

Example 10.5 Using Definition 10.3, show that $f(n) = 3n^2 + 6n + 7$ is $\Omega(n^2)$. In other words, $f(n)$ has at least quadratic growth.

Solution: Let $g(n) = n^2$. We need to find constants $c > 0$ and k such that $f(n) \geq c \cdot g(n)$ for all $n \geq k$.

We can write:

$$f(n) = 3n^2 + 6n + 7 \geq 3n^2.$$

Here, $c = 3$ and $k = 1$ satisfy the inequality, so we conclude $f(n) = \Omega(n^2)$. ◀

The following properties hold for Ω notation:

- If $f(n) = \Omega(g(n))$ and $g(n) = \Omega(h(n))$, then $f(n) = \Omega(h(n))$.
- Constant functions are $\Omega(1)$, meaning they have a lower bound that does not depend on input size.

In formal terms, $T(n) = \Omega(g(n))$ if there exist constants $c > 0$ and $k \geq 0$ such that $T(n) \geq c \cdot g(n)$ for all $n \geq k$.

For example, if a sorting algorithm always requires a minimum of $n \log n$ comparisons, we might say it has a time complexity of $\Omega(n \log n)$.

Example 10.6

Suppose $f(n)$ is defined as follows.

$$f(n) = \begin{cases} 2n^{2.1}\log^3 n & \text{if } n \bmod 3 = 0 \\ 10n^2 \log^{50} n & \text{if } n \bmod 3 = 1 \\ 6n^2 \log^{30} n^{80} & \text{if } n \bmod 3 = 2 \end{cases}$$

Provide a Big-$\mathcal{O}$ upper bound and Big-Ω lower bound for $f(n)$.

Solution: To find a Big-$\mathcal{O}$ upper bound and a Big-Ω lower bound for the function we examine each case and choose the term that grows the fastest as $n \to \infty$ for the upper bound, and the term that grows the slowest for the lower bound.

Determine the Big-$\mathcal{O}$ Upper Bound

To find the Big-$\mathcal{O}$ upper bound, we look for the term with the highest growth rate as $n \to \infty$. The term $2n^{2.1}\log^3 n$ grows slightly faster than n^2 terms, as $n^{2.1}\log^3 n$ will dominate $n^2\log^{50} n$ and $n^2\log^{30} n^{30}$ as n becomes large.

Thus, a Big-$\mathcal{O}$ upper bound for $f(n)$ is:

$$f(n) = \mathcal{O}(n^{2.1}\log^3 n).$$

Determine the Big-Ω Lower Bound

For the Big-Ω lower bound, we look for the term with the slowest growth rate as $n \to \infty$. The term $n^2\log^{50} n$ grows slower than $n^{2.1}\log^3 n$, and it also has a slower growth rate than $n^2\log^{30} n^{80}$.

Therefore, a Big-Ω lower bound for $f(n)$ is:

$$f(n) = \Omega(n^2\log^{50} n).$$

In conclusion, we have:

$$f(n) = \mathcal{O}(n^{2.1}\log^3 n) \quad \text{and} \quad f(n) = \Omega(n^2\log^{50} n).$$

◀

To summarise:

- Big-$\mathcal{O}$ notation provides an upper bound, describing the *worst-case* scenario.
- Omega (Ω) notation gives a lower bound, capturing the *best-case* scenario.
- Theta (Θ) notation describes a tight bound, representing the algorithm's *consistent* complexity.

By understanding these notations, we gain a fuller perspective on an algorithm's efficiency across different scenarios.

10.4 Further Analysis

Not all functions can be easily classified using Big-$\mathcal{O}$, Omega (Ω), and Theta (Θ) notation. In some cases, the growth rate of a function may not be easily determined, or the function may not fit neatly into one of these categories. In such cases, we can use additional tools to analyse the growth rate of functions more precisely.

Little-o Notation

Little-o notation provides a *strict upper bound* on the growth rate of a function. It describes a scenario where the function grows slower than another function, but not necessarily at the same rate. This notation is useful for indicating that an algorithm's time complexity is strictly less than a given function.

Definition 10.4 (Little-o)

$f(n)$ is $o(g(n))$ if for any positive constant $\epsilon > 0$, there exists a constant $k \geq 0$ such that for all $n \geq k$,

$$f(n) < \epsilon \cdot g(n)$$

This means that $f(n)$ grows strictly slower than $g(n)$ as n becomes sufficiently large. ♣

The phrase "n becomes sufficiently large" indicates that the above inequality does not need to hold for all $n \geq 0$, but only for $n \geq k$ for some constant k.

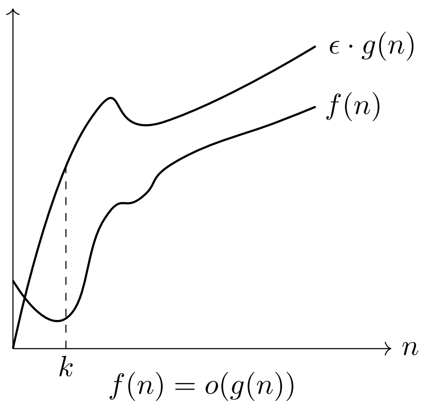


Figure 10.7: o-notation: $f(n)$ is strictly bounded above by $\epsilon \cdot g(n)$ for large n.

Figure 10.7 shows the concept of Little-o notation, illustrating that $f(n)$ is strictly bounded above by $\epsilon \cdot g(n)$. This notation provides a strict upper bound on the running time of an algorithm, describing the asymptotic growth rate of the algorithm's runtime in a more precise manner. Or stated more succinctly:

$$f(n) = o(g(n)) : \quad f \text{ grows strictly slower than } g$$

Example 10.7 Using Definition 10.4, show that $f(n) = 3n^2 + 6n + 7$ is $o(n^3)$. In other words, $f(n)$ grows strictly slower than n^3.

Solution: Let $g(n) = n^3$. We need to show that for any positive constant $\epsilon > 0$, there exists a constant k such that $f(n) < \epsilon \cdot g(n)$ for all $n \geq k$.

We can write:

$$f(n) = 3n^2 + 6n + 7 < \epsilon \cdot n^3$$

for sufficiently large n. Dividing both sides by n^3, we get:

$$\frac{f(n)}{n^3} = \frac{3n^2+6n+7}{n^3} = \frac{3}{n} + \frac{6}{n^2} + \frac{7}{n^3}$$

As $n \to \infty$, the right-hand side approaches 0, which is less than any positive constant ϵ. Therefore, $f(n) = o(n^3)$. ◀

This leads us to the following proposition

Proposition 10.2

Let $f(n)$ and $g(n)$ be two nonnegative functions, and suppose

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$$

Then $f(n) = o(g(n))$, meaning $f(n)$ grows strictly slower than $g(n)$ as $n \to \infty$. ♠

Example 10.8 Use Proposition 10.2 to prove that if $f(n) = 3n^2 + 6n + 7$, then $f(n) = o(n^3)$. In other words, $f(n)$ grows strictly slower than n^3.

Solution:

Let $g(n) = n^3$. According to Proposition 10.2, we need to compute the limit:

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{3n^2+6n+7}{n^3}$$

Dividing each term in the numerator by n^3, we get:

$$\lim_{n\to\infty} \frac{3n^2}{n^3} + \frac{6n}{n^3} + \frac{7}{n^3} = \lim_{n\to\infty} \left(\frac{3}{n} + \frac{6}{n^2} + \frac{7}{n^3}\right)$$

Now, as $n \to \infty$, the terms $\frac{3}{n}$, $\frac{6}{n^2}$, and $\frac{7}{n^3}$ approach 0, so we have:

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$$

Since this limit is 0, by Proposition 10.2, it follows that $f(n) = o(n^3)$.

Thus, we have shown that $f(n) = 3n^2 + 6n + 7$ grows strictly slower than n^3, as required. ◀

The following properties hold for o notation:

- If $f(n) = o(g(n))$, then $g(n) \neq o(f(n))$.
- If $f(n) = o(g(n))$ and $g(n) = o(h(n))$, then $f(n) = o(h(n))$.

Little-o notation provides a more precise way to describe the growth rate of functions, especially when the function grows strictly slower than another function. This notation is useful for indicating that an algorithm's time complexity is strictly less than a given function, providing a more detailed understanding of the algorithm's complexity.

Little-ω Notation

Little-ω notation provides a *strict lower bound* on the growth rate of a function. It describes a scenario where the function grows faster than another function, but not necessarily at the same rate. This notation is useful for indicating that an algorithm's time complexity is strictly greater than a given function.

Definition 10.5 (Little-ω)

$f(n)$ is $\omega(g(n))$ if for any positive constant $\epsilon > 0$, there exists a constant $k \geq 0$ such that for all $n \geq k$,

$$f(n) > \epsilon \cdot g(n)$$

This means that $f(n)$ grows strictly faster than $g(n)$ as n becomes sufficiently large. ♣

The phrase "n becomes sufficiently large" indicates that the above inequality does not need to hold for all $n \geq 0$, but only for $n \geq k$ for some constant k.

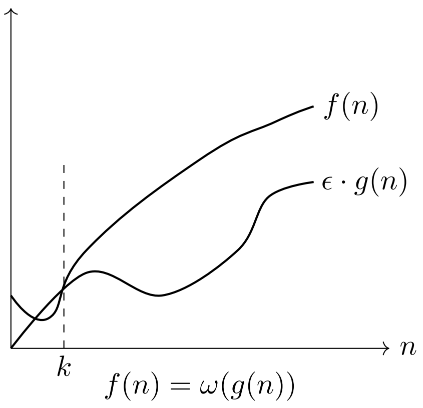


Figure 10.8: ω-notation: $f(n)$ is strictly bounded below by $\epsilon \cdot g(n)$ for large n.

Figure 10.8 shows the concept of Little-ω notation, illustrating that $f(n)$ is strictly bounded below by $\epsilon \cdot g(n)$. This notation provides a strict lower bound on the running time of an algorithm, describing the asymptotic growth rate of the algorithm's runtime in a more precise manner. Or stated more succinctly:

$$f(n) = \omega(g(n)) : \quad f \text{ grows strictly faster than } g$$

Example 10.9 Using Definition 10.5, show that $f(n) = n^3 + 6n + 7$ is $\omega(n^2)$. In other words, $f(n)$ grows strictly faster than n^2.

Solution: Let $g(n) = n^2$. We need to show that for any positive constant $\epsilon > 0$, there exists a constant k such that $f(n) > \epsilon \cdot g(n)$ for all $n \geq k$.

We can write:

$$f(n) = n^3 + 6n + 7 > \epsilon \cdot n^2$$

for sufficiently large n. Dividing both sides by n^2, we get:

$$\frac{f(n)}{n^2} = \frac{n^3 + 6n + 7}{n^2} = n + \frac{6}{n} + \frac{7}{n^2}$$

As $n \to \infty$, the right-hand side approaches n, which is greater than any positive constant ϵ. Therefore, $f(n) = \omega(n^2)$. ◀

This leads us to the following proposition:

Proposition 10.3

Let $f(n)$ and $g(n)$ be two nonnegative functions, and suppose

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty$$

Then $f(n) = \omega(g(n))$, meaning $f(n)$ grows strictly faster than $g(n)$ as $n \to \infty$. ♠

Example 10.10 Use Proposition 10.3 to prove that if $f(n) = n^3 + 6n + 7$, then $f(n) = \omega(n^2)$. In other words, $f(n)$ grows strictly faster than n^2.

Solution: Let $g(n) = n^2$. According to Proposition 10.3, we need to compute the limit:

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{n^3 + 6n + 7}{n^2}$$

Dividing each term in the numerator by n^2, we get:

$$\lim_{n\to\infty} \frac{n^3}{n^2} + \frac{6n}{n^2} + \frac{7}{n^2} = \lim_{n\to\infty} \left(n + \frac{6}{n} + \frac{7}{n^2}\right)$$

Now, as $n \to \infty$, the term n approaches ∞, so we have:

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty$$

Since this limit is ∞, by Proposition 10.3, it follows that $f(n) = \omega(n^2)$.

Thus, we have shown that $f(n) = n^3 + 6n + 7$ grows strictly faster than n^2, as required. ◀

The following properties hold for ω notation:

- If $f(n) = \omega(g(n))$, then $g(n) \neq \omega(f(n))$.
- If $f(n) = \omega(g(n))$ and $g(n) = \omega(h(n))$, then $f(n) = \omega(h(n))$.

Little-ω notation provides a more precise way to describe the growth rate of functions, especially when the function grows strictly faster than another function. This notation is useful for indicating that an algorithm's time complexity is strictly greater than a given function, providing a more detailed understanding of the algorithm's complexity.

Using Little-o and Little-ω to find $\mathcal{O}$ and Ω

Little-o and Little-ω notations can be used to provide more precise bounds when determining Big-$\mathcal{O}$ and Ω notations. They help in identifying whether a function grows strictly slower or faster than another function, which can be useful in asymptotic analysis.

Example 10.11 Consider the function $f(n) = n^2 + 3n + 2$. We want to determine the Big-$\mathcal{O}$ and Ω notations.

1. **Big-$\mathcal{O}$ Notation:**
   To determine Big-$\mathcal{O}$, we consider the ratio $\frac{f(n)}{g(n)}$, where $g(n) = n^2$. Using Proposition 10.4, we compute

the $\limsup$[3]:

$$\limsup_{n\to\infty} \frac{f(n)}{g(n)} = \limsup_{n\to\infty} \frac{n^2+3n+2}{n^2} = \limsup_{n\to\infty} \left(1+\frac{3}{n}+\frac{2}{n^2}\right).$$

As $n \to \infty$, the terms $\frac{3}{n}$ and $\frac{2}{n^2}$ approach 0, so:

$$\limsup_{n\to\infty} \frac{f(n)}{g(n)} = 1.$$

Since this value is finite, we conclude that $f(n) = \mathcal{O}(n^2)$.

2. **Big-Ω Notation:**

   To determine Big-Ω, we compute the $\liminf$ of the same ratio[4]:

$$\liminf_{n\to\infty} \frac{f(n)}{g(n)} = \liminf_{n\to\infty} \frac{n^2+3n+2}{n^2} = \liminf_{n\to\infty} \left(1+\frac{3}{n}+\frac{2}{n^2}\right).$$

   Again, as $n \to \infty$, the terms $\frac{3}{n}$ and $\frac{2}{n^2}$ approach 0, so:

$$\liminf_{n\to\infty} \frac{f(n)}{g(n)} = 1.$$

   Since this value is positive, we conclude that $f(n) = \Omega(n^2)$.

In conclusion, we have:

$$f(n) = \mathcal{O}(n^2) \quad \text{and} \quad f(n) = \Omega(n^2).$$

By using Little-o and Little-ω notations, we can provide more precise bounds for Big-$\mathcal{O}$ and Ω notations, helping us better understand the growth rates of functions and the complexity of algorithms. But perhaps more importantly, we can use the concept of limits to determine Big-$\mathcal{O}$ and Ω notations from Little-o and Little-ω notations.

This leads us to the following propositions:

Proposition 10.4

Let $f(n)$ and $g(n)$ be two nonnegative functions. If

$$\limsup_{n\to\infty} \frac{f(n)}{g(n)} < \infty$$

then $f(n) = \mathcal{O}(g(n))$, meaning $f(n)$ grows at most as fast as $g(n)$ asymptotically.

And also:

Proposition 10.5

Let $f(n)$ and $g(n)$ be two nonnegative functions. If

$$\liminf_{n\to\infty} \frac{f(n)}{g(n)} > 0$$

then $f(n) = \Omega(g(n))$, meaning $f(n)$ grows at least as fast as $g(n)$ asymptotically.

These propositions provide a more formal way to determine Big-$\mathcal{O}$ and Ω notations from Little-o and Little-ω notations. By examining the limits of the ratios of functions, we can determine whether a function

[3]The $\limsup$ notation stands for the "limit superior" and considers the upper bound of the values of the ratio $\frac{f(n)}{g(n)}$ as n grows. Unlike a regular limit, which may not exist if the function oscillates, $\limsup$ will always exist for real-valued sequences and captures the maximum growth behaviour in the long run.

[4]The $\liminf$ notation stands for "limit inferior" and considers the lower bound of the values of the ratio $\frac{f(n)}{g(n)}$ as n grows. Unlike a standard limit, which may fail to exist if the function oscillates, $\liminf$ always exists for real-valued sequences. It captures the minimum long-term growth behaviour, highlighting the slowest rate at which $f(n)$ grows relative to $g(n)$ as n becomes large.

grows at most as fast or at least as fast as another function, providing a more rigorous analysis of growth rates.

Example 10.12 Use Proposition 10.5 to prove that if $f(n) = 3n^2 + 5n + 7$, then $f(n) = \Omega(n^2)$.

Solution: Let $g(n) = n^2$. We need to compute the lower limit (lim inf) of $\frac{f(n)}{g(n)}$ as n approaches infinity:

$$\liminf_{n\to\infty} \frac{f(n)}{g(n)} = \liminf_{n\to\infty} \frac{3n^2+5n+7}{n^2}.$$

Dividing each term in the numerator by n^2, we have:

$$= \liminf_{n\to\infty} \left(3 + \frac{5}{n} + \frac{7}{n^2}\right).$$

As $n \to \infty$, the terms $\frac{5}{n}$ and $\frac{7}{n^2}$ approach 0. Thus, we obtain:

$$\liminf_{n\to\infty} \frac{f(n)}{g(n)} = 3.$$

Since this limit is a positive constant $c = 3 > 0$, by Proposition 10.5, it follows that $f(n) = \Omega(n^2)$. This example demonstrates that $f(n) = 3n^2 + 5n + 7$ has quadratic growth in the lower bound sense. ◀

Example 10.13 Determine if $f(n) = 2n\sin(n) + 3n$ is $\mathcal{O}(n)$ and $\Omega(n)$ by using the limits as described in Proposition 10.4.

Solution: Let $g(n) = n$. We will examine the behavior of $\frac{f(n)}{g(n)}$ as $n \to \infty$.

First, express $\frac{f(n)}{g(n)}$ as follows:

$$\frac{f(n)}{g(n)} = \frac{2n\sin(n)+3n}{n} = 2\sin(n) + 3.$$

Now, analyze the limit behavior as $n \to \infty$:

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 2\sin(n) + 3.$$

Since $\sin(n)$ oscillates between -1 and 1 as n grows, the expression $2\sin(n) + 3$ fluctuates between 1 and 5. Therefore, the simple limit $\lim_{n\to\infty} \frac{f(n)}{g(n)}$ does not exist due to this oscillation.

However, we can compute the $\limsup$ and $\liminf$ of $\frac{f(n)}{g(n)}$:

$$\limsup_{n\to\infty} \frac{f(n)}{g(n)} = 5 \quad \text{and} \quad \liminf_{n\to\infty} \frac{f(n)}{g(n)} = 1.$$

Since $\limsup$ is finite (5), we conclude that $f(n) = \mathcal{O}(n)$ by Proposition 10.4. Similarly, because $\liminf$ is positive (1), we conclude that $f(n) = \Omega(n)$ by Proposition 10.5. ◀

Using Limits to Determine Asymptotic Notations (An Informal Approach)

An informal but practical approach to determine Big-$\mathcal{O}$ and Ω involves examining the ratio $\frac{f(n)}{g(n)}$ as $n \to \infty$. We have already seen this approach in Propositions 10.1-10.5. While not as rigorous as formal propositions, this method provides quick insights into growth rates.

The following rules illustrate how limits can be applied to each notation:

Informal Limit Approach

$$\lim_{n\to\infty}\frac{f(n)}{g(n)} \neq \infty \quad \Rightarrow f(n)=\mathcal{O}(g(n))$$
$$\lim_{n\to\infty}\frac{f(n)}{g(n)} \neq 0 \quad \Rightarrow f(n)=\Omega(g(n))$$

Note: This approach is an informal guideline and not a strict mathematical definition. There are cases where this limit-based method may not apply, but it often provides a convenient tool for determining asymptotic behavior.

These rules work as follows:

- **Big-$\mathcal{O}$:** If the limit exists and is not infinity, then $f(n)=\mathcal{O}(g(n))$, meaning that $f(n)$ does not grow faster than $g(n)$.
- **Big-Ω:** If the limit exists and is non-zero, then $f(n)=\Omega(g(n))$, meaning that $f(n)$ grows at least as fast as $g(n)$.

Example 10.14 Determine if $\sqrt{n}\cdot\log n=\mathcal{O}(n\log n)$ using the informal limit approach.

Solution: We need to compute the limit of the ratio $\frac{\sqrt{n}\cdot\log n}{n\log n}$ as $n\to\infty$. Simplifying the expression, we get:

$$\lim_{n\to\infty}\frac{\sqrt{n}\cdot\log n}{n\log n}=\lim_{n\to\infty}\frac{\sqrt{n}}{n}=\lim_{n\to\infty}\frac{1}{\sqrt{n}}=0.$$

Since the limit is finite, we conclude that $\sqrt{n}\cdot\log n=\mathcal{O}(n\log n)$. We also notice that since the limit is 0, we can infer that $\sqrt{n}\cdot\log n\neq\Omega(n\log n)$.

◀

Solution: To determine if $\sqrt{n}\cdot\log n=\mathcal{O}(n\log n)$, we compute the following limit:

$$\lim_{n\to\infty}\frac{\sqrt{n}\cdot\log n}{n\log n}=\lim_{n\to\infty}\frac{\sqrt{n}}{n}=\lim_{n\to\infty}\frac{1}{\sqrt{n}}.$$

As $n\to\infty$, $\frac{1}{\sqrt{n}}$ approaches 0. Since the limit is finite (not infinity), this result implies that $\sqrt{n}\cdot\log n=\mathcal{O}(n\log n)$.

Therefore, we conclude that $\sqrt{n}\cdot\log n=\mathcal{O}(n\log n)$. ◀

The limit approach, while convenient, is not definitive. For some functions (e.g., oscillating functions), the formal definitions and propositions should be applied directly. This limit-based technique serves as a useful heuristic, particularly when the asymptotic growth is clear.

Comparing Notations

We conclude this chapter by comparing the different asymptotic notations and their relationships:

1. $f(n)=\mathcal{O}(g(n))$ and $f(n)=\Omega(g(n))\Leftrightarrow f(n)=\Theta(g(n))$
   This relationship states that if $f(n)$ is both asymptotically bounded above by $g(n)$ (Big-$\mathcal{O}$) and bounded below by $g(n)$ (Big-Ω), then $f(n)$ and $g(n)$ grow at the same rate, denoted by $\Theta(g(n))$. In other words, $f(n)$ grows neither faster nor slower than $g(n)$ asymptotically, as $n\to\infty$.

2. $f(n) = o(g(n)) \Rightarrow f(n) = \mathcal{O}(g(n))$
   If $f(n) = o(g(n))$, this means that $f(n)$ grows strictly slower than $g(n)$ as $n \to \infty$. Therefore, $f(n)$ is asymptotically bounded above by $g(n)$ but does not grow as fast, justifying that $f(n) = \mathcal{O}(g(n))$.
3. $f(n) = \mathcal{O}(g(n)) \Leftrightarrow g(n) = \Omega(f(n))$
   This equivalence shows the symmetric nature of Big-$\mathcal{O}$ and Big-Ω notations. If $f(n)$ is asymptotically bounded above by $g(n)$ (meaning $f(n)$ grows no faster than $g(n)$), then we can also say that $g(n)$ grows at least as fast as $f(n)$ (i.e., $g(n) = \Omega(f(n))$).
4. $f(n) = \omega(g(n)) \Rightarrow f(n) = \Omega(g(n))$
   When $f(n) = \omega(g(n))$, this means that $f(n)$ grows strictly faster than $g(n)$ as $n \to \infty$. Since $f(n)$ is asymptotically larger, it follows that $f(n)$ grows at least as fast as $g(n)$, making $f(n) = \Omega(g(n))$ true.
5. $f(n) = o(g(n)) \Leftrightarrow g(n) = \omega(f(n))$
   This equivalence indicates the symmetric nature of Little-o and Little-ω notations. If $f(n) = o(g(n))$, meaning $f(n)$ grows strictly slower than $g(n)$, then we can also say that $g(n)$ grows strictly faster than $f(n)$, or $g(n) = \omega(f(n))$. This relationship highlights the strict difference in their growth rates as $n \to \infty$.

In this chapter, we examined time complexity to evaluate algorithm efficiency. We introduced Big-$\mathcal{O}$, Omega (Ω), and Theta (Θ) notations as ways to establish upper, lower, and tight bounds on growth rates, which help us understand how runtime and space requirements scale with input size.

We highlighted the importance of ignoring constants and lower-order terms in asymptotic analysis, focusing instead on the dominant term. Additionally, we explored Little-o and Little-ω notations for stricter upper and lower bounds, providing more precise insights into complexity.

Finally, we compared these notations and their key relationships, building a foundation for evaluating algorithm complexity both theoretically and practically — preparing us for hands-on time complexity analysis in the next chapter.

Chapter 11 Code Analysis Techniques

In software engineering, understanding the efficiency of code is pivotal for building scalable, responsive applications. Building on the principles of asymptotic notation, we now turn to practical code analysis — dissecting how code components like loops, conditionals, and sequential statements contribute to an algorithm's overall complexity.

Code analysis can be broadly divided into two categories:

1. **Static Analysis:** This approach involves examining code without executing it. We focus on evaluating the structure and operations of the code, particularly through complexity analysis. This is the primary focus of this chapter.
2. **Dynamic Analysis:** By running code on various inputs, we measure real execution times and memory usage in specific environments. While valuable, dynamic analysis is outside the scope of this chapter and is typically used for performance profiling.

Asymptotic notation offers a high-level view of an algorithm's performance, predicting its growth rate as the input size increases. However, theoretical complexity doesn't account for constants, smaller terms, or machine-specific factors. In practical scenarios, these details can substantially impact runtime, especially for applications handling frequent operations on large datasets.

Consider two algorithms, both with a time complexity of $\mathcal{O}(n)$. While theoretically equivalent, one algorithm may have a constant factor several times larger than the other, causing it to run more slowly in practice. A real-world understanding of complexity requires analysing each part of the code, estimating the number of operations in concrete terms, and identifying areas for potential optimization.

Below you see two algorithms for filtering even numbers from an integer array. Both algorithms have $\mathcal{O}(n)$ complexity, but one uses a single loop to achieve the task, while the other uses multiple loops, effectively doubling the work and thus introducing a larger constant factor.

Algorithm 11.1: Single pass to filter even numbers

```java
public List<Integer> filterEvenSinglePass(int[] arr) {
    List<Integer> evenNumbers = new ArrayList<>();
    for (int i = 0; i < arr.length; i++) {
        if (arr[i] % 2 == 0) {
            evenNumbers.add(arr[i]);
        }
    }
    return evenNumbers;
}
```

Algorithm 11.2: Multiple passes to filter even numbers

```java
public List<Integer> filterEvenMultiplePasses(int[] arr) {
    List<Integer> evenNumbers = new ArrayList<>();

    // First pass: count even numbers
    int evenCount = 0;
    for (int i = 0; i < arr.length; i++) {
        if (arr[i] % 2 == 0) {
            evenCount++;
        }
    }

    // Second pass: collect even numbers
    for (int i = 0; i < arr.length; i++) {
        if (arr[i] % 2 == 0) {
            evenNumbers.add(arr[i]);
        }
    }

    return evenNumbers;
}
```

Algorithm 11.1 uses a single pass through the array to identify even numbers. As it iterates, it directly appends each even number to a result list, resulting in a single $\mathcal{O}(n)$ traversal.

Algorithm 11.2, however, makes two full passes through the array. The first pass counts the number of even elements, and the second pass collects these even elements into a list. This structure still results in $\mathcal{O}(n)$ complexity but requires $2n$ operations instead of n, effectively doubling the work.

In practical terms, Algorithm 11.1 is faster because it completes the task in a single traversal, while Algorithm 11.2 introduces a larger constant factor by making two full passes through the array. This example highlights how even within the same $\mathcal{O}(n)$ complexity, different implementations can have varying real-world performance due to differing constant factors.

In this chapter, we will systematically analyse code components, beginning with pseudocode to abstract language-specific syntax. We will then explore the complexity of various code structures, including loops, conditionals, and sequences. Finally, we will consider practical aspects, discussing how constants and lower-order terms, often overlooked in asymptotic notation, can affect real-world performance.

11.1 Pseudocode

To analyse algorithms effectively without getting bogged down by language-specific syntax, we use pseudocode — a simplified, high-level representation of code that emphasizes the logic and structure of an algorithm. Pseudocode allows us to describe and analyse algorithms in a way that's independent of any programming language, focusing purely on the sequence and efficiency of operations. Writing pseudocode serves as a bridge between conceptual algorithm design and implementation. By abstracting away language details, pseudocode helps in visualizing the core operations of an algorithm, making it easier to identify complexity and optimize performance before implementing it in code.

Pseudocode also enables collaborative work, allowing team members with varying language expertise to understand and contribute to algorithm design without needing to know specific syntax. Moreover, pseudocode is useful for documenting algorithms, as it provides a concise, yet clear summary of the steps involved, which is easier to read than fully detailed code.

We follow the conventions from Cormen et al., 2022 that adhere to a specific set of formatting rules to ensure clarity and uniformity:

1. **Loop Structures**: We use `for` and `while` to denote iterations:
   - `for` $i = 1$ `to` n represents a loop incrementing i by 1 from 1 through n.
   - `while` condition represents a loop that continues while the specified condition is true.
2. **Assignment and Operations**: The assignment operator $=$ indicates variable assignment, e.g., $sum = 0$ sets `sum` to zero. Basic arithmetic operations (`+, -, *, /`) follow standard notation.
3. **Procedure and Function Calls**: Procedures are named in small caps, such as SUM-ARRAY(arr), while variables and identifiers appear in italics.
4. **Comments**: Comments appear in a Bordeaux colour, providing contextual information for each step.

To illustrate, let's look at a simple example of an algorithm that calculates the sum of elements in an array. Here, we use pseudocode to describe the algorithm, focusing on the logic and steps involved.

```
SUM-ARRAY($arr$)
$sum = 0$                          // initialise sum to zero
for $i = 0$ to $arr.length - 1$
    $sum = sum + arr[i]$           // Add the current element to sum
return $sum$                       // Return the final sum
```

In this pseudocode:

- The procedure SUM-ARRAY takes an array `arr` as input.
- We initialise a variable `sum` to zero.
- A `for` loop iterates over each element in the array, adding it to `sum`.
- Finally, the function returns the computed `sum`.

This pseudocode captures the algorithm's essential steps without syntax-specific details, making it easy to analyse and adapt. Here is a more complex example:

```
FIND-MAX-EVEN-SUM($arr$)
maxSum = 0                                // Initialise max sum to zero
currentSum = 0                            // Track the sum of the current subarray
for $i = 0$ to $arr.length - 1$
    if $arr[i] \bmod 2 == 0$
        $currentSum = currentSum + arr[i]$           // Add even number to current sum
        $maxSum = \text{MAX}(maxSum, currentSum)$    // Update if currentSum > maxSum
    else
        $currentSum = 0$                  // Reset current sum if odd number is encountered
return $maxSum$                           // Return the maximum sum of any contiguous even subarray
```

This procedure calculates the maximum sum of contiguous even elements in an array:

- The procedure takes an input array `arr` and initialises `maxSum` to track the maximum sum and `currentSum` to accumulate the sum of the current contiguous even subarray.
- A single **`for`** loop iterates through `arr`:
- If an element is even, it is added to `currentSum`, and `maxSum` is updated if `currentSum` exceeds it.
- If an element is odd, `currentSum` is reset to 0, as the contiguous subarray is interrupted.

Here are two more examples of pseudocode, one for *iterative binary search* and another for *recursive binary search*. The pseudocode for both algorithms is similar, but the recursive version uses a function call to invoke itself, while the iterative version uses a loop to manage the search process.

```
BINARY-SEARCH-ITERATIVE($arr, key$)
$low = 0$                                 // Initialise low index
$high = arr.length - 1$                   // Initialise high index
while $low \leq high$
    $mid = \lfloor (low + high)/2 \rfloor$    // Calculate mid index
    if $arr[mid] == key$
        return $mid$                      // Key found at mid index
    elseif $arr[mid] < key$
        $low = mid + 1$                   // Update low index
    else
        $high = mid - 1$                  // Update high index
return $-1$                               // Key not found
```

Here's a step-by-step explanation of what the code does:

1. **Initialisation**:
   - `low` is initialised to the first index of the array `A`.
   - `high` is initialised to the last index of the array `A`.
2. **While Loop**:
   - The loop continues as long as `low` is less than or equal to `high`.
3. **Calculate Mid-Index**:

- `mid` is calculated as the floor value of the average of `low` and `high`.

4. **Comparison**:
   - If the element at the `mid` index is equal to the `key`, the function returns `mid`, indicating the position of the `key` in the array.
   - If the element at the `mid` index is less than the `key`, it means the `key` must be in the right half of the array. Therefore, `low` is updated to `mid + 1`.
   - If the element at the `mid` index is greater than the `key`, it means the `key` must be in the left half of the array. Therefore, `high` is updated to `mid - 1`.
5. **Key Not Found**:
   - If the loop exits without finding the `key`, the function returns `-1`, indicating that the `key` is not present in the array.

This algorithm efficiently searches for a `key` in a sorted array by repeatedly dividing the search interval in half. The time complexity of binary search is $\mathcal{O}(\log n)$, making it much faster than a linear search for large arrays. The algorithm is **iterative** because it uses a **`while`** loop to repeatedly execute a block of code until a certain condition is met. In this case, the loop continues to execute as long as `low` is less than or equal to `high`. Each iteration of the loop performs a comparison and updates the `low` and `high` indices accordingly. This iterative approach contrasts with a recursive approach, where the function would call itself with updated parameters until the base condition is met.

```
BINARY-SEARCH-RECURSIVE($arr, key, low, high$)
if $low > high$
    return $-1$                      // Key not found
$mid = \lfloor (low + high)/2 \rfloor$          // Calculate mid index
if $arr[mid] == key$
    return $mid$                     // Key found at mid index
elseif $arr[mid] < key$
    // Search right half
    return BINARY-SEARCH-RECURSIVE($arr, key, mid + 1, high$)
else
    // Search left half
    return BINARY-SEARCH-RECURSIVE($arr, key, low, mid - 1$)
```

This code implements a recursive binary search algorithm. The function BINARY-SEARCH-RECURSIVE takes an array `arr`, a `key` to search for, and the `low` and `high` indices that define the current search range.

1. **Base Case**: If `low` is greater than `high`, the function returns -1, indicating that the key is not found in the array.
2. **Calculate Midpoint**: The midpoint `mid` is calculated as the floor of the average of `low` and `high`.
3. **Key Comparison**:
   - If the element at `mid` index is equal to the `key`, the function returns `mid`, indicating the key is found at this index.
   - If the element at `mid` index is less than the `key`, the function recursively searches the right half of the array by updating `low` to `mid + 1`.

- If the element at `mid` index is greater than the `key`, the function recursively searches the left half of the array by updating `high` to `mid - 1`.

This algorithm efficiently searches for a `key` in a sorted array by repeatedly dividing the search interval in half. The algorithm is **recursive** because it calls itself with updated parameters to narrow down the search range. In each recursive call, the function calculates the midpoint of the current range and performs a comparison. Depending on the result, the function either returns the `mid` index if the key is found, or recursively calls itself on either the left or right half of the current interval, adjusting the `low` or `high` index accordingly. The recursion continues until the base condition is met, either finding the key or determining that it is not present. This recursive approach contrasts with an iterative approach, which uses a loop to perform the search rather than recursive function calls.

11.2 Construct Analysis

Construct analysis involves breaking down code into its fundamental components, such as loops, conditionals, and sequences, to understand how each contributes to the overall complexity. By analysing these constructs, we can identify bottlenecks, inefficiencies, and areas for optimization within an algorithm. This section will explore the complexity of various code structures, focusing on loops, conditionals, and sequences, and how they impact an algorithm's performance.

Basic Operations

The time required by a function or procedure is proportional to the number of "basic operations" that it performs. Examples of basic operations include:

1. **Arithmetic Operation**: A single arithmetic operation, such as addition (+), multiplication (*), subtraction (-), or division (/).
2. **Assignment**: Assigning a value to a variable, e.g., $x = 0$.
3. **Test/Comparison**: Evaluating a condition, such as $x == 0$ or $i \leq n$.
4. **Function Call**: Invoking a function or procedure, e.g., calling FIND-MAX-EVEN-SUM.
5. **Return Statement**: Returning a value from a function, e.g., `return total`.
6. **Read Operation**: Reading a value of a primitive type (e.g., integer, float, character, boolean) from input or memory.
7. **Write Operation**: Writing a value of a primitive type to output or memory.

These basic operations form the foundation of more complex operations, and each adds one unit time unit to the construct they are part of, i.e. $\mathcal{O}(1)$ time complexity. By counting the number of basic operations within a construct, we can estimate the time complexity of that construct. We will **only include 1-5** in our analysis and omit read and write operations as they are not as common in algorithm analysis.

All of the following are $\mathcal{O}(1)$ algorithms because they perform a constant number of basic operations:

```
INC($x$)
1  return $x + 1$
```

```
MUL$(x, y)$
return $x \times y$
```

```
FOO$(x)$
$y = x \times 77.3$
return $x/8.2$
```

```
BAR$(x, y)$
$z = x + y$
$w = x \times y$
$q = (w^z) \bmod 870$
return $9 \times q$
```

In the last example, the function BAR performs a series of arithmetic operations, assignments, and a modulo operation. Each of these operations is considered a basic operation, and the function's time complexity is $\mathcal{O}(1)$ because it performs a constant number of basic operations. Here is break-down:

- **Line 1:** $z = x + y$
  - **Arithmetic Operation**: 1 (addition)
  - **Assignment**: 1 (assigning to z)

  Total Operations: 2
- **Line 2:** $w = x \times y$
  - **Arithmetic Operation**: 1 (multiplication)
  - **Assignment**: 1 (assigning to w)

  Total Operations: 2
- **Line 3:** $q = (w^z) \bmod 870$
  - **Arithmetic Operations**: 2 (one for exponentiation w^z and one for modulo 870)
  - **Assignment**: 1 (assigning to q)

  Total Operations: 3
- **Line 4: return** $9 \times q$
  - **Arithmetic Operation**: 1 (multiplication by 9)
  - **Assignment (Implicit)**: 1 (returning the result)

  Total Operations: 2

Adding up the operations for each line, we get:

$$2 + 2 + 3 + 2 = 9$$

The total number of basic operations is constant (9 operations), making this an $O(1)$ algorithm.

Sequential Statements

When an algorithm contains multiple statements that are executed one after the other, they are referred to as "sequential statements" or "consecutive statements" The total time complexity of sequential statements depends on the individual complexities of each of those statements. Specifically, the overall complexity is dominated by the highest-order term among the individual statements.

Consider a sequence of statements:

```
statement 1;
statement 2;
    ⋮
statement k;
```

The total time for executing these statements can be represented as:

$$\text{Total time} = \text{time(statement 1)} + \text{time(statement 2)} + \cdots + \text{time(statement } k\text{)}$$

If each statement is "simple," meaning it only involves basic operations, then each of these statements takes a constant amount of time, resulting in a total complexity of $\mathcal{O}(1)$. This applies to cases where the number of operations within each statement does not depend on the size of the input - we saw this in the previous section.

However, in more general scenarios, the complexity of sequential statements is determined by the highest order among the individual complexities. For example, if one statement has a complexity of $\mathcal{O}(n)$ and another has a complexity of $\mathcal{O}(n^2)$, the overall complexity is $\mathcal{O}(n^2)$, as the highest-order term will dominate for sufficiently large input sizes.

As we have seen multiple times, in asymptotic analysis, lower-order terms become insignificant as the input size grows larger, and hence are not included in the final representation of the algorithm's complexity. Thus, when evaluating sequential statements, only the term with the highest growth rate needs to be considered for the purposes of estimating the total time complexity.

Simple Loops

Loops are primary determinants of an algorithm's complexity, as they often define how many times a particular operation is executed. A simple loop that runs from 1 to n has a time complexity of $\mathcal{O}(n)$. This is because the loop executes n times, and if the sequence of statements within the loop has a complexity of $\mathcal{O}(1)$, the total time complexity for the loop is $n \times \mathcal{O}(1)$, which simplifies to $\mathcal{O}(n)$.

The general form of a loop can be represented as:

```
for $i$ in $1, 2, \dots, N$ loop
  sequence of statements
end loop;
```

Here, the loop executes N times, which means that the sequence of statements also executes N times. Assuming that each of these statements is a basic operation with complexity $\mathcal{O}(1)$, the total time complexity of the loop is $\mathcal{O}(N)$.

An illustrative example involves considering a function that performs multiplication without using the built-in multiplication operator (*). Instead, the function uses a loop to add the first operand repeatedly:

```
MUL2($x, y$)
result = 0
for $i = 0$ to $y - 1$
    $result = result + x$
return $result$
```

In this function, the loop runs y times, performing a constant-time addition operation in each iteration. Hence, the complexity of this function is $\mathcal{O}(y)$. The time complexity here depends directly on the size of the input y, demonstrating how loop bounds directly affect the performance characteristics of an algorithm.

Functions containing simple for loops that iterate through the entire input generally have a complexity of $\mathcal{O}(n)$, where n is the size of the input. This is because each element of the input is processed in sequence, leading to a linear relationship between the input size and the number of operations performed.

Consider the following:

```
FACTORIAL($n$)
$result = 1$
for $num = 1$ to $n$
    $result = result \times num$
return $result$
```

The given code for calculating the factorial of a number n involves a single loop that iterates from 1 to n. For each iteration, it performs a constant-time multiplication operation (result *= num). Thus, the loop runs n times, and each iteration is an $\mathcal{O}(1)$ operation.

Therefore, the overall time complexity of the FACTORIAL function is $\mathcal{O}(n)$. Now consider:

```
FACTORIAL2($n$)
$result = 1$
$count = 0$
for $num = 1$ to $n$
    $result = result \times num$
    $count = count + 1$
return $result$
```

The function FACTORIAL2 is very similar to the previous example of calculating the factorial of a number n. It contains a single loop that iterates from 1 to n. During each iteration, it performs two constant-time operations: multiplication (result *= num) and addition (count += 1). Since these operations are both $\mathcal{O}(1)$ and the loop runs n times, the total time complexity is still $\mathcal{O}(n)$.

The presence of the additional assignment (count += 1) does not change the *asymptotic* complexity, as it is still a constant-time operation repeated n times. Thus, the overall complexity remains $\mathcal{O}(n)$. The additional operation may however affect the practical performance of the function, as it increases the number of basic operations executed in each iteration.

If-Then-Else Statements

For an `if-then-else` structure, the time complexity depends on the time required to evaluate the condition and the time taken by the branch that executes. The overall time for an `if-then-else` statement is the *maximum* of the time complexities of each branch plus the time to evaluate the condition, as only one branch executes.

- **If the condition is simple**, like comparing two numbers, it takes constant time, $\mathcal{O}(1)$.
- **If the condition is more complex**, such as checking membership in a large data structure, the time complexity will reflect that of the condition evaluation.

Consider the following pseudocode for an `if-then-else` statement:

```
if cond
    block 1 (sequence of statements)
else
    block 2 (sequence of statements)

```

Here, either block 1 will execute, or block 2 will execute. Therefore, the worst-case time is the slower of the two possibilities:

$$\max(\text{time(block 1)}, \text{time(block 2)})$$

If block 1 takes $\mathcal{O}(1)$ and block 2 takes $\mathcal{O}(N)$, the overall time complexity for the `if-then-else` statement is $\mathcal{O}(N)$.

The complexity of conditionals depends on the condition itself. Condition evaluation time can be constant, linear, or even worse, depending on the specific check being performed.

Consider the following examples to illustrate this further:

```
COUNT_TS($a\_str$)
count = 0
for each $char$ in $a\_str$
    if $char$ = 't'
        $count = count + 1$
return $count$
```

In this example, we used an `if` statement to check if one character is equal to another, which is a constant-time operation. The overall function runs in linear time with respect to the size of `a_str`, since the condition check is simple. Letting $n = |\texttt{a\_str}|$, this function has time complexity $\mathcal{O}(n)$.

Now, consider this code:

```
COUNT_SAME_LTRS($a\_str, b\_str$)
count = 0
for each $char$ in $a\_str$
    if $char \in b\_str$
        $count = count + 1$
return $count$
```

This code looks similar to COUNT_TS but the conditional check `char ∈ b_str` is more complex. In the worst case, it requires checking every character in `b_str`. Big-O notation captures the worst-case scenario, so we ask: what input could maximize the number of steps? Here, the worst case occurs when `char` is not in `b_str`, as each character in `b_str` must be checked before returning `False`.

Let $n = |\texttt{a\_str}|$ and $m = |\texttt{b\_str}|$. The **`for`** loop iterates $\mathcal{O}(n)$ times, and each iteration performs an $\mathcal{O}(m)$ conditional check. Since we execute an $\mathcal{O}(m)$ check $\mathcal{O}(n)$ times, the overall complexity of this function is $\mathcal{O}(nm)$.

While Loops

While loops are fundamental constructs for creating iterative processes in algorithms. Unlike **`for`** loops, which iterate a predetermined number of times, a **`while`** loop continues executing as long as a given condition remains true. This flexibility makes **`while`** loops suitable for scenarios where the number of iterations is not known in advance and depends on runtime conditions.

The structure of a **`while`** loop can be represented as:

```
while condition is true:
    sequence of statements
```

- The condition is checked before each iteration. If it is false initially, the loop body will not execute.
- The number of iterations depends on how the condition evolves and transitions to false.

The time complexity of a **`while`** loop depends on the complexity of the condition check and the number of iterations required for the condition to become false.

Consider the following example:

```
FACTORIAL3($n$)
$result = 1$
while $n > 0$
    $result = result \times n$
    $n = n - 1$
return $result$
```

In this example, the loop continues while `n > 0`. Since `n` is decremented by 1 during each iteration, the loop runs n times, resulting in a time complexity of $\mathcal{O}(n)$.

Consider another example that uses a `while` loop to split a string:

```
CHAR-SPLIT($a\_str$)
1  $result = [\,]$ // initialise an empty list
2  $index = 0$
3  while $len(a\_str) \neq len(result)$
4      $append(result, a\_str[index])$
5      $index = index + 1$
6  return $result$
```

In this function, `len(a_str)` and `len(result)` are constant-time operations, as are string indexing and list appending. The while loop runs until the length of `result` matches the length of `a_str`, resulting in n iterations where n is the length of the input string. Hence, the overall time complexity of this function is $\mathcal{O}(n)$.

Consider this example:

```
SUM-UNTIL-THRESHOLD($arr, threshold$)
1  $sum = 0$
2  $i = 0$
3  while $sum < threshold$ and $i < arr.length$
4      $sum = sum + arr[i]$
5      $i = i + 1$
6  return $sum$
```

Here, the loop runs until the condition `sum < threshold` is no longer true or until all elements in the array `arr` have been processed. The time complexity of this loop depends on both the length of the array and the value of `threshold`. In the worst case, if `threshold` is very large, the loop will iterate over every element in the array, resulting in a complexity of $\mathcal{O}(n)$, where n is the length of the array.

In general, `while` loops are powerful constructs for iterating over conditions that are not known in advance. The key to analysing their complexity lies in understanding how the condition evolves with each iteration and estimating the number of iterations needed to reach termination. Proper analysis involves understanding both the evolution of the condition and the cost of each iteration.

Nested Loops

Nested loops are an important construct in algorithm design, where one loop runs inside another. This type of structure significantly affects the time complexity of an algorithm due to the multiplicative effect of multiple iterations.

Consider the general form of a nested loop:

```
for $i = 1$ to $n$ :
    sequence of statements
    for $j = 1$ to $m$ :
        sequence of statements
```

In the case of nested loops, the inner loop executes fully for each iteration of the outer loop. This means that the total number of iterations is the product of the number of iterations of each loop. If the outer loop runs n times and the inner loop runs m times, the total number of times the body of the inner loop executes is $n \times m$. If m and n are both proportional to the size of the input, then the time complexity of the nested loop is $\mathcal{O}(n \times m)$.

If both the inner and outer loops iterate over the same range, say from 1 to n, then the total time complexity of the nested loop structure is $\mathcal{O}(n^2)$. This is often the case when performing operations on a two-dimensional structure, such as a matrix, where each row and column must be iterated over.

For example:

```
SUM-MATRIX($matrix$)
sum = 0
for $i = 0$ to $matrix.rows - 1$
    for $j = 0$ to $matrix.cols - 1$
        $sum = sum + matrix[i][j]$
return $sum$
```

In this pseudocode, the outer loop iterates over each row of the matrix, while the inner loop iterates over each column. If the matrix has n rows and m columns, the time complexity of this nested loop is $\mathcal{O}(n \times m)$. In the case of a square matrix, where $n = m$, the complexity becomes $\mathcal{O}(n^2)$.

In some cases, the number of iterations of the inner loop depends on the index of the outer loop. A common example is a triangular loop structure, where the inner loop runs fewer times as the outer loop progresses.

Consider the following pseudocode:

```
TRIANGULAR-SUM($n$)
$sum = 0$
for $i = 1$ to $n$
    for $j = i$ to $n$
        $sum = sum + (i + j)$
return $sum$
```

Here, the inner loop starts from the current value of i and runs until n. This means that, on the first iteration, the inner loop runs n times, on the second iteration it runs $n - 1$ times, and so on, until it runs only once. The total number of iterations for the nested loops can be calculated as (see Appendix A):

$$\sum_{i=1}^{n}(n-i+1) = \frac{n(n+1)}{2}$$

Thus, the time complexity of this nested loop is $\mathcal{O}(n^2)$. The triangular structure reduces the number of total iterations compared to a full $\mathcal{O}(n^2)$ loop, but the complexity remains quadratic.

In the case of more than two nested loops, the same principle applies. If there are k nested loops, each running n times, the overall time complexity is $\mathcal{O}(n^k)$.

When analysing nested loops, it is crucial to understand the range and bounds of each loop. If the bounds are dynamic and depend on the value of other variables, the complexity can be harder to determine and may require a more in-depth analysis of how those bounds change during execution.

Nested loops are often used for problems involving multi-dimensional data structures or for comparing each element in a collection against every other element (e.g., sorting algorithms like Bubble Sort, which has a time complexity of $\mathcal{O}(n^2)$ due to its nested loop structure).

Recursion

This book, along with the courses it supports, does not cover the topic of recursion in depth. Here we discuss recursion briefly in the context of construct analysis simply for the sake of completeness. We have already seen recursion in the context of binary search. Recursion is a powerful tool in algorithm design, allowing functions to call themselves to solve problems by breaking them down into smaller sub-problems. Recursive algorithms are especially effective for problems that have a natural hierarchical or repetitive structure, such as tree traversal, divide-and-conquer algorithms, or calculating factorials.

A key part of any recursive function is the **base case** — the condition under which the function stops calling itself to prevent infinite recursion.

Consider the general structure of a recursive function:

- **Base Case**: The termination condition that stops the recursion.
- **Recursive Case**: The part of the function that calls itself with a modified input, progressing towards the base case.

A classic example of a recursive function is the calculation of a factorial:

$$\texttt{factorial}(n) = \begin{cases} 1 & \text{if } n = 0 \\ n \times \texttt{factorial}(n-1) & \text{if } n > 0 \end{cases}$$

In pseudocode, the factorial function can be represented as:

```
FACTORIAL4($n$)
1  if $n == 0$
2      return $1$
3  return $n \times$ FACTORIAL($n - 1$)
```

In this pseudocode:

- The **base case** occurs when $n == 0$, returning 1.
- The **recursive case** multiplies n by the result of $\textsc{Factorial4}(n-1)$.

The time complexity of a recursive function depends on how many times the function calls itself and how much work is done at each level of recursion. For the factorial function:

- The function makes a single recursive call in each step, decrementing n by 1 each time.
- The recursion depth is n, and each call performs $\mathcal{O}(1)$ work (a multiplication).

Thus, the time complexity of the factorial function is $\mathcal{O}(n)$.

To determine the time complexity of a recursive algorithm, we can use recurrence relations. A recurrence relation expresses the time complexity of a recursive function in terms of its sub-problems. For example, the recurrence relation for the factorial function can be represented as:

$$T(n) = T(n-1) + \mathcal{O}(1)$$

This indicates that the time to compute `factorial`(n) is equal to the time to compute `factorial`$(n-1)$ plus a constant amount of work, leading to a complexity of $\mathcal{O}(n)$.

Another well-known example of a recursive function is the Fibonacci sequence:

$$\texttt{fibonacci}(n) = \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ \texttt{fibonacci}(n-1) + \texttt{fibonacci}(n-2) & \text{if } n > 1 \end{cases}$$

The corresponding pseudocode is:

```
NAIVE-FIBONACCI($n$)
1  if $n == 0$
2      return $0$
3  if $n == 1$
4      return $1$
5  return FIBONACCI($n-1$) + FIBONACCI($n-2$)
```

In this example:

- The function calls itself twice in the recursive case, creating a branching structure.
- The **time complexity** is $\mathcal{O}(2^n)$, which grows exponentially due to repeated sub-problems being solved multiple times. This highlights the inefficiency of the naive recursive Fibonacci implementation.

To improve the efficiency of recursive functions that solve overlapping sub-problems, such as the Fibonacci sequence, **memoization** can be used. Memoization involves storing the results of sub-problems to avoid redundant calculations.

For example, applying memoization to the Fibonacci function changes the time complexity from $\mathcal{O}(2^n)$ to $\mathcal{O}(n)$ by ensuring each sub-problem is solved only once.

Recursion can often be replaced by iteration, especially for problems where the recursive depth may become a concern. For example, the factorial function can be rewritten iteratively which we saw earlier. Here is the iterative version of the factorial function:

```
FACTORIAL-ITERATIVE($n$)
    $result = 1$
    for $i = 1$ to $n$
        $result = result \times i$
    return $result$
```

This iterative approach has the same time complexity, $\mathcal{O}(n)$, as the recursive version but avoids the overhead of recursive calls and is not limited by the system's stack size.

Tail recursion is a special form of recursion where the recursive call is the last operation in the function. In some languages, tail-recursive functions can be optimized by the compiler into iteration, preventing additional stack usage. Consider the following tail-recursive factorial:

```
FACTORIAL-TAIL($n, accumulator$)
    if $n == 0$
        return $accumulator$
    return FACTORIAL-TAIL($n - 1, accumulator \times n$)
```

Here, the recursive call is in **tail position**, meaning no further computation is needed after the call returns. Tail recursion helps reduce the risk of *stack overflow* and is often optimized into a loop by compilers.

Recursion is a versatile tool in algorithm design, enabling elegant solutions to complex problems by breaking them down into smaller, more manageable sub-problems. However, it comes with trade-offs in terms of space and time complexity, especially with deep or exponential recursion.

Analysing recursive algorithms involves understanding the recurrence relations that define their behaviour, identifying base cases, and estimating how many times the function will be called. While recursion can lead to clear and concise code, an iterative approach or optimization techniques like memoization and tail call optimization are often needed to improve efficiency.

11.3 Loop Analysis

In the previous section, we already discussed the time complexity of loops. This section will focus on analysing loops in more detail, including nested loops, loops with variable bounds, and loops with conditional exits.

When asked for the time complexity in terms of Big-$\mathcal{O}$ notation, you are always asked to find the tightest upper bound, by which we mean the one that grows the *slowest*. So while $\mathcal{O}(n^2)$ is an upper bound for $f(n) = 4n + 3$, it is **not** the *tightest* upper bound. The tightest upper bound is $O(n)$.

Techniques for Practical Loop Analysis

Having understood the different types of loops and their general contributions to complexity, it is now time to delve into practical techniques for analysing these constructs in real-world scenarios. This section will provide actionable methods that readers can apply to systematically evaluate loops in their algorithms, focusing on both theoretical and practical aspects of analysis.

Establishing Loop Boundaries

To determine the time complexity of a loop, start by establishing the boundaries of each iteration. This involves identifying the starting point, ending point, and increment pattern for loop variables. Loop boundaries define the upper limit of the iteration count, allowing us to deduce the worst-case scenario for the number of times a loop will execute.

For example, consider a loop of the form:

for $i = a$ to b by c :

The number of iterations in this loop can be calculated as:

$$\left(\frac{b-a}{c}\right)+1$$

where a is the starting value, b is the ending value, and c is the step size. Analyzing these boundaries helps determine the overall complexity of the loop.

The following examples illustrate this concept.

Example 11.1

```
LOOP-CONSTANT-INCREMENT($n$)
1  for $i = 1$ to $n$
2      // Perform some $\mathcal{O}(1)$ operation
```

In this example, the loop runs from a to b with a step size of 1. The number of iterations is:

$$\left(\frac{b-a}{1}\right)+1=(b-a+1)=(n-1+1)=n$$

In this example, i starts at 1 and increments by 1 until n. The loop executes n times, leading to a time complexity of $\mathcal{O}(n)$.

The above example directly illustrates the use of the formula for calculating iteration counts when the step size is constant. Here, $a = 1$, $b = n$, and $c = 1$, resulting in n iterations.

Example 11.2

```
LOOP-GEOMETRIC-GROWTH($n$)
1  $i = 1$
2  while $i \leq n$
3      // Perform some $\mathcal{O}(1)$ operation
4      $i = 2 \times i$
```

In this example, i starts at 1 and doubles on each iteration until it exceeds n. The number of iterations can be determined as follows:

The value of i evolves as:

$$1, 2, 4, 8, \ldots, 2^k$$

where $2^k \leq n$. To determine the number of iterations, we solve:

$$2^k \leq n \implies k \leq \log_2(n)$$

Thus, the number of iterations is approximately $\log_2(n)$, leading to a time complexity of:

$$\mathcal{O}(\log n)$$

In this example, the increment pattern is multiplicative rather than additive. Although the formula given earlier directly applies to additive increments, the principle of analyzing loop boundaries to determine iteration count still holds. Here, we determine how many times we can double i before it exceeds n, resulting in a logarithmic complexity.

Example 11.3

```
LOOP-DECREMENT($n$)
1  $i = n$
2  while $i > 0$
3      // Perform some $\mathcal{O}(1)$ operation
4      $i = i - 3$
```

In this example, i starts at n and decreases by 3 each time until it becomes less than or equal to 0. The number of iterations is given by:

$$i : n, (n-3), (n-6), \ldots$$

The loop continues until $i \leq 0$. To determine the number of iterations, we need to consider how many times we can subtract 3 from n:

$$\frac{n}{3}$$

Thus, the number of iterations is approximately:

$$\left(\frac{n}{3}\right)$$

The overall time complexity of the loop is:

$$\mathcal{O}(n)$$

In this example, the step size is -3. Using the general formula provided earlier, we have $a = n$, $b = 0$, and $c = -3$. This leads to an iteration count of approximately $n/3$. The principle of establishing boundaries and using the formula directly applies here, highlighting how the decrement affects the number of iterations.

Complexity Through Summation

In algorithms with multiple consecutive loops, the overall complexity is the sum of the complexities of each loop. However, when loops are nested, the complexity is the product of their individual complexities. For instance, two consecutive loops each with $\mathcal{O}(n)$ complexity still result in $\mathcal{O}(n)$, whereas a nested structure results in $\mathcal{O}(n^2)$. Understanding this distinction is critical.

Nested loops often require evaluating the total number of operations through summation. If the inner loop is dependent on the value of the outer loop, it is helpful to write down the summation that represents the number of iterations for all loops. For instance:

```
for $i = 1$ to $n$ :
  for $j = 1$ to $i$ :
```

The number of total iterations is given by the summation (see Appendix A):

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2} = \mathcal{O}(n^2)$$

This type of analysis is particularly useful for nested loops with dependent ranges. The following examples illustrate this concept.

Example 11.4

```
LINEAR-SUM-LARGE-CONSTANT($n$)
1  for $i = 1$ to $n$
2      for $j = 1$ to $10^9$
3          // Perform some $\mathcal{O}(1)$ operation
```

In this example, the outer loop runs from 1 to n, while the inner loop runs a constant number of times (from 1 to 10^9). The total number of iterations is:

$$\sum_{i=1}^{n} 10^9 = 10^9 \times n = \mathcal{O}(n)$$

Although the inner loop has a very large constant number of iterations, the overall complexity remains linear in n, as the constant factor does not affect the asymptotic growth.

Example 11.5

```
QUADRATIC-SUM($n$)
1  for $i = 1$ to $n$
2      for $j = 1$ to $i$
3          // Perform some $\mathcal{O}(1)$ operation
```

In this example, the outer loop runs from 1 to n, and the inner loop runs from 1 to i. The total number of iterations is:

$$\sum_{i=1}^{n}\sum_{j=1}^{i} 1 = \sum_{i=1}^{n} i = \frac{n(n+1)}{2} = \mathcal{O}(n^2)$$

In this expression:

- The outer summation, $\sum_{i=1}^{n}$, represents the iterations of the outer loop over i.
- The inner summation, $\sum_{j=1}^{i}$, represents the iterations of the inner loop for each fixed i.

The 1 in the inner summation represents performing a constant-time operation on each inner loop iteration. This expression counts each inner loop iteration individually for all pairs of i and j that satisfy $1 \leq j \leq i$. This double summation is equivalent to $\sum_{i=1}^{n} = i$, since, for each i, the inner loop runs exactly i times. To see why the singular sum is equivalent to the double sum, let us try to implement the single sum:

```
SUM-UP-TO($n$)
total = 0
for $i$ = 1 to $n$
    total = total + $i$
return total
```

In this code:

- The outer loop runs from 1 to n.
- On each iteration, the current value of i is added to `total`.
- By the end of the loop, `total` will contain the sum $1 + 2 + \cdots + n$, which is exactly $\sum_{i=1}^{n} i$.

While the sum $\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$ grows proportionally to n^2, the time complexity of the SUM-UP-TO(N) function remains linear, $\mathcal{O}(n)$, because it performs a constant-time operation in each of its n iterations. This demonstrates that an algorithm can have an output magnitude that grows quadratically with n, yet its time complexity depends on the number of operations, which increases linearly.

Example 11.6

```
LOGARITHMIC-GROWTH($n$)
for $i$ = 1 to $n$
    $j$ = 1
    while $j < n$
        // Perform some $\mathcal{O}(1)$ operation
        $j = 2 \times j$
```

In this example, the outer loop runs from 1 to n, while the inner loop starts at 1 and doubles j until it reaches or exceeds n. The number of iterations for the inner loop is:

$$\sum_{i=1}^{n} \log_2(n) = n \cdot \mathcal{O}(\log n) = \mathcal{O}(n \log n)$$

Thus, the overall time complexity of this nested structure is $\mathcal{O}(n \log n)$.

Worst-Case, Best-Case, and Average-Case Complexity

Not all loops run the same number of times in all scenarios. The best-case, worst-case, and average-case behaviours can vary based on the condition that controls the loop's termination. For instance, if a loop exits early due to a condition being met, the best-case complexity could be significantly less than the worst-case complexity. Understanding these different cases and how they affect runtime is vital for real-world applications where inputs may vary.

Example 11.7

```
FIND-TARGET($arr, target$)
1  $n = arr.length$
2  for $i = 1$ to $n$
3      if $arr[i] == target$
4          return $i$ // Return the index of the target if found
```

In this example, the goal is to find a target element in an array of length n. The number of iterations the loop will run depends on the position of the target:

- **Best-Case Complexity:** The target is found at the first position in the array. The loop runs only once, resulting in a complexity of $\boldsymbol{\Theta(1)}$.[1]
- **Worst-Case Complexity:** The target is not in the array or is found at the last position. The loop runs n times, resulting in a complexity of $\boldsymbol{\mathcal{O}(n)}$.
- **Average-Case Complexity:** On average, the target may be located somewhere in the middle of the array. Assuming uniform distribution, the average number of iterations is approximately $n/2$, leading to an average-case complexity of $\boldsymbol{\Theta(n)}$.

This example illustrates how the same loop can exhibit different behaviours depending on the input, resulting in distinct best, worst, and average-case complexities.

Example 11.8

Consider the iterative binary search algorithm mentioned previously in this chapter applied to a sorted array of length n. Binary search is a divide-and-conquer algorithm that works by repeatedly dividing the search interval in half.

[1]One may think that the lower bound is $\Omega(1)$. In the best case, when the target is at the first position, the algorithm performs a constant amount of work. The time complexity is both $\boldsymbol{\mathcal{O}(1)}$ (upper bound) and $\boldsymbol{\Omega(1)}$ (lower bound), which means it's $\boldsymbol{\Theta(1)}$ (a tight bound). Using $\boldsymbol{\Theta(1)}$ is more precise because it indicates the time is bounded both above and below by a constant.

```
BINARY-SEARCH-ITERATIVE($arr, key$)
low = 0                          // Initialise low index
high = arr.length − 1            // Initialise high index
while low ≤ high
    mid = ⌊(low + high)/2⌋       // Calculate mid index
    if arr[mid] == key
        return mid               // Key found at mid index
    elseif arr[mid] < key
        low = mid + 1            // Update low index
    else
        high = mid − 1           // Update high index
return −1                        // Key not found
```

- **Best-Case Complexity:** In the best-case scenario, the target element is located at the middle of the array on the very first comparison. Since this takes only one operation, the complexity is $\boldsymbol{\Theta(1)}$. This notation is used because it provides a tight bound — the algorithm always takes constant time in the best case.
- **Worst-Case Complexity:** In the worst-case scenario, the algorithm keeps dividing the search interval in half until it reaches an empty interval, which takes $\log n$ steps. The complexity is therefore $\boldsymbol{\mathcal{O}(\log n)}$, as this represents an upper bound on the time the algorithm takes, ensuring it will never exceed this value.
- **Average-Case Complexity:** On average, assuming the target is equally likely to be at any position, the expected number of comparisons is proportional to the logarithm of n. Therefore, the average-case complexity is $\boldsymbol{\Omega(\log n)}$. This notation represents a lower bound for the average runtime, indicating that at least $\log n$ operations are required on average.

Example 11.9

Consider the Naïve String Matching algorithm applied to a text string T of length n and a pattern string P of length m. The goal is to find all occurrences of P within T by checking for a match at every possible shift.

```
NAIVE-STRING-MATCHING($T, P$)
n = T.length
m = P.length
for s = 0 to n − m
    i = 0
    while i < m and T[s + i] == P[i]
        i = i + 1
    if i == m
        report s // Pattern found at position s
```

- **Best-Case Complexity:** In the best-case scenario, the pattern and the text share no common characters at the positions being compared. At each shift, the first character comparison fails, leading to a total of $n - m + 1$ comparisons. Thus, the complexity is $\boldsymbol{\Omega(n)}$, representing a lower bound on the time the algorithm takes.

- **Worst-Case Complexity:** In the worst-case scenario, the text and pattern contain repeated sequences causing maximum overlap during comparisons. The algorithm performs up to m comparisons at each of the $n - m + 1$ shifts, resulting in a complexity of $\boldsymbol{\mathcal{O}(nm)}$. This notation represents an upper bound, ensuring the algorithm will not exceed this time.
- **Average-Case Complexity:** Assuming the characters in the text and pattern are randomly distributed, the expected number of comparisons per shift is constant. Therefore, the average-case complexity is $\boldsymbol{\Theta(n)}$, providing a tight bound on the expected running time.

This example illustrates how the use of Ω, O, and Θ provides precise information about the complexity for different scenarios of an algorithm's execution.

Examples of Loop Analysis

We now provide a series of examples to illustrate how to analyse loops.

Example 11.10

Consider the following algorithm:

```
LOOP1($n$)
1  $i = 1$
2  while $i \leq n$
3      $i = 3 \times i$
```

At each iteration of the `while` loop, the variable i is multiplied by 3. We want to determine how many times the loop will execute before i exceeds n.

Let k be the number of iterations the loop executes.

Initially, $i_0 = 1$.

After the first iteration:

$$i_1 = 3 \times i_0 = 3 \times 1 = 3$$

After the second iteration:

$$i_2 = 3 \times i_1 = 3 \times 3 = 3^2 = 9$$

After k iterations:

$$i_k = 3^k \times i_0 = 3^k$$

The loop continues as long as $i_k \leq n$. Therefore, the condition for loop termination is:

$$3^k > n$$

Taking the logarithm base 3 of both sides:

$$k > \log_3 n$$

Since k must be an integer, the number of iterations is:

$$k = \lceil \log_3 n \rceil$$

Conclusion: The loop runs $\lceil \log_3 n \rceil$ times, so the time complexity of the algorithm is $\boldsymbol{\mathcal{O}(\log n)}$.

Example 11.11

Consider the following algorithm:

```
LOOP2($n$)
1  $s = 1$
2  for $i = 1$ to $n$
3      for $j = 1$ to $n$
4          $s = s + 1$
```

We have two nested `for` loops:

- The outer loop runs from $i = 1$ to $i = n$, executing n iterations.
- The inner loop runs from $j = 1$ to $j = n$, also executing n iterations for each iteration of the outer loop.

At each iteration of the inner loop, a constant-time operation is performed: $s = s + 1$.

Total number of iterations:

- The inner loop runs n times for each of the n iterations of the outer loop.
- Therefore, the total number of times $s = s + 1$ is executed is:
  $$n \times n = n^2$$

Conclusion: The algorithm performs around n^2 operations, so the time complexity is $\boldsymbol{\mathcal{O}(n^2)}$.

Example 11.12

Consider the following algorithm:

```
LOOP3($n$)
1  $i = 1$
2  while $i \leq n$
3      $i = 2 \times i$
```

At each iteration of the `while` loop, the variable i is multiplied by 2. We need to determine how many times the loop will execute before i exceeds n.

Let k be the number of iterations the loop executes.

- **Initialisation:** $i_0 = 1$.
- **Iteration Update:** After each iteration, i is updated as:
  $$i_k = 2^k \times i_0 = 2^k$$
- **Termination Condition:** The loop continues while $i_k \leq n$. Thus:
  $$2^k \leq n$$

- **Solving for k:** Taking the logarithm base 2 of both sides:
  $k \le \log_2 n$
- **Number of Iterations:** Since k must be an integer, the loop executes:
  $k = \lfloor \log_2 n \rfloor + 1$

Conclusion: The loop runs $\lfloor \log_2 n \rfloor + 1$ times, so the time complexity of the algorithm is $\boldsymbol{\mathcal{O}(\log n)}$.

Example 11.13

Consider the following algorithm:

```
LOOP4($n$)
1  $i = 1$
2  while $i \le n \times n$
3      $i = 3 \times i$
```

At each iteration of the `while` loop, the variable i is multiplied by 3. We need to determine how many times the loop will execute before i exceeds n^2.

Let k be the number of iterations the loop executes.

- **Initialisation:** $i_0 = 1$.
- **Iteration Update:** After each iteration:
  $i_k = 3^k \times i_0 = 3^k$
- **Termination Condition:** The loop continues while $i_k \le n^2$. Thus:
  $3^k \le n^2$
- **Solving for k:** Taking the logarithm base 3 of both sides:
  $k \le \log_3 n^2$
- **Simplifying:** Since $\log_3 n^2 = 2\log_3 n$:
  $k \le 2\log_3 n$
- **Number of Iterations:** The loop executes:
  $k = \lfloor 2\log_3 n \rfloor + 1$

Conclusion: The loop runs $\lfloor 2\log_3 n \rfloor + 1$ times, so the time complexity of the algorithm is $\boldsymbol{\mathcal{O}(\log n)}$.

Example 11.14

Consider the following algorithm:

```
LOOP5($n$)
1  $i = 1$
2  while $i \le n$
3      $j = 0$
4      while $j \le n$
5          $j = j + 1$
6      $i = 2 \times i$
```

We need to determine the total number of times the inner statement $j = j + 1$ is executed.

- **Outer Loop (`while` $i \le n$):**
  - The variable i starts at 1 and is doubled each time: $i = 1, 2, 4, 8, \dots$.
  - The number of iterations of the outer loop is:
    $$k = \lfloor \log_2 n \rfloor + 1$$
  - This is because the loop continues while $i \le n$, and i grows exponentially.
- **Inner Loop (`while` $j \le n$):**
  - For each iteration of the outer loop, j is initialised to 0.
  - The inner loop increments j from 0 to n, executing $n + 1$ times.
- **Total Operations:**
  - The inner loop runs $n + 1$ times per outer loop iteration.
  - Total number of times $j = j + 1$ is executed:
    $$(\lfloor \log_2 n \rfloor + 1) \times (n + 1)$$
  - Simplifying, we have:
    $$\mathcal{O}((\log n) \times n) = \mathcal{O}(n \log n)$$

Conclusion: The algorithm performs $(\lfloor \log_2 n \rfloor + 1) \times (n+1)$ operations, so the time complexity is $\boldsymbol{\mathcal{O}(n \log n)}$.

Example 11.15

Consider the following algorithm:

```
LOOP6($n$)
1  $i = 1$
2  while $i \le n$
3      $j = i$
4      while $j \le n$
5          $j = j + 1$
6      $i = 2 \times i$
```

Complete Time Complexity Analysis:

We need to determine the total number of times the inner statement $j = j + 1$ is executed.

- **Outer Loop (`while` $i \le n$):**
  - The variable i starts at 1 and is doubled each time: $i = 1, 2, 4, 8, \dots$.
  - The number of iterations of the outer loop is:
    $$k_{\max} = \lfloor \log_2 n \rfloor + 1$$
  - This is because the loop continues while $i \le n$, and i grows exponentially.
- **Inner Loop (`while` $j \le n$):**
  - For each iteration of the outer loop, j is initialised to the current value of i.
  - The inner loop increments j from i to n, executing $n - i + 1$ times.
- **Total Operations:**

- The total number of times $j = j + 1$ is executed is:
  $$\text{Total} = \sum_{k=0}^{k_{\max}-1} (n - i_k + 1)$$
  where $i_k = 2^k$.
- Substituting i_k:
  $$\text{Total} = \sum_{k=0}^{\lfloor \log_2 n \rfloor} (n - 2^k + 1)$$
- Simplify the sum:
  $$\begin{aligned}\text{Total} &= \sum_{k=0}^{\lfloor \log_2 n \rfloor} (n + 1 - 2^k) \\ &= (\lfloor \log_2 n \rfloor + 1)(n+1) - \sum_{k=0}^{\lfloor \log_2 n \rfloor} 2^k\end{aligned}$$
- Calculate the sum of 2^k:
  $$\sum_{k=0}^{\lfloor \log_2 n \rfloor} 2^k = 2^{\lfloor \log_2 n \rfloor + 1} - 1 \leq 2^{\log_2 n + 1} - 1 = 2n - 1$$
- Therefore:
  $$\text{Total} \leq (\log_2 n + 1)(n+1) - (2n - 1)$$
- Simplify:
  $$\begin{aligned}\text{Total} &\leq (\log_2 n + 1)(n+1) - 2n + 1 \\ &= (\log_2 n)(n+1) + (n+1) - 2n + 1 \\ &= (\log_2 n)(n+1) - n + 2\end{aligned}$$
- For large n, the term $(\log_2 n)(n+1)$ dominates, so:
  $$\text{Total} = \mathcal{O}(n \log n)$$

Conclusion: The algorithm performs $(\log_2 n)(n+1) - n + 2$ operations, so the time complexity is $\boldsymbol{\mathcal{O}(n \log n)}$.

Simplified Time Complexity Analysis:

We can simplify the analysis by examining the behaviour of the outer and inner loops separately.

- **Outer Loop (`while` $i \leq n$):**
  - The variable i starts at 1 and is doubled each time:
    $$i = 1, 2, 4, 8, \ldots, n$$
  - This doubling behaviour results in a logarithmic number of iterations:
    $$\text{Number of iterations of the outer loop} = \mathcal{O}(\log n)$$
- **Inner Loop (`while` $j \leq n$):**
  - For each value of i, j starts at i and increments by 1 until $j > n$.
  - The number of iterations of the inner loop for a given i is:
    $$n - i + 1 \leq n$$
  - Which means the inner loop runs approximately n times for each value of i.
  - Therefore, the inner loop runs $\mathcal{O}(n)$ times for each iteration of the outer loop.
- **Total Complexity:**
  - The outer loop runs $\mathcal{O}(\log n)$ times.
  - For each iteration of the outer loop, the inner loop runs $\mathcal{O}(n)$ times.

- Therefore, the total number of operations is:

  $$\mathcal{O}(\log n) \times \mathcal{O}(n) = \mathcal{O}(n \log n)$$

11.4 Code Analysis

We now provide a series of examples to illustrate how to analyse code snippets. All code will be Java code, but the principles apply to any programming language.

Example 11.16

Algorithm 11.3: Simple Function

```java
public static int sumSquareDifference(int n) {
    int sum = (n * (n + 1)) / 2;
    int sumOfSquares = (n * (n + 1) * (2 * n + 1)) / 6;
    return (sum * sum) - sumOfSquares;
}
```

Line 2: `int sum = (n * (n + 1)) / 2;`

- **Assignments**:
  - Assign sum = ... : **1 time unit**
- **Arithmetic Operations**:
  - Multiplication `n * (n + 1)`: **1 time unit**
  - Addition `n + 1`: **1 time unit**
  - Division `(n * (n + 1)) / 2`: **1 time unit**
- **Total for Line 2**: **4 time units**

Line 3: `int sumOfSquares = (n * (n + 1) * (2 * n + 1)) / 6;`

- **Assignments**:
  - Assign `sumOfSquares` = ... : **1 time unit**
- **Arithmetic Operations**:
  - Multiplication `n * (n + 1)`: **1 time unit**
  - Addition `n + 1`: **1 time unit**
  - Multiplication `(n * (n + 1)) * (2 * n + 1)`: **1 time unit**
  - Multiplication `2 * n`: **1 time unit**
  - Addition `2 * n + 1`: **1 time unit**
  - Division `(...) / 6`: **1 time unit**
- **Total for Line 3**: **7 time units**

Line 4: `return (sum * sum) - sumOfSquares;`

- **Arithmetic Operations**:
  - Multiplication `sum * sum`: **1 time unit**
  - Subtraction `(sum * sum) - sumOfSquares`: **1 time unit**
- **Return Operation** (write to output or return value):
  - Return value: **1 time unit**

- **Total for Line 4**: **3 time units**

Summary of Total Time (T):

- **Line 1**: 4 time units
- **Line 2**: 7 time units
- **Line 3**: 3 time units

Thus, the total time T to execute the function `sumSquareDifference` is:

$T = 4 + 7 + 3 = 14$ time units

This detailed analysis allows us to precisely calculate the time T based on counting all individual operations, and we see that the time complexity is $\mathcal{O}(\mathbf{1})$.

We could also place the analysis as code comments in the function to document the time complexity of each line.

Algorithm 11.4: Simple Function

```java
    public static int sumSquareDifference(int n)
    {
        int sum = (n * (n + 1)) / 2;     // 4 time units
        int sumOfSquares = (n * (n + 1) * (2 * n + 1)) / 6; // 7 time units
        return (sum * sum) - sumOfSquares;  // 3 time units:
    } // Total time: T(n) = 4 + 7 + 14 time units so we get O(1)
```

We will prefer the latter method as it provides a clear and concise way to document the time complexity of the function or method inside the code itself.

Example 11.17

In the following example, we present an algorithm to perform the

Algorithm 11.5: Binary Search Iterative

```java
public static int binarySearchIter(int[] a, int x) {
    int low = 0, high = a.length - 1;

    while (low <= high) {
        int mid = (low + high) / 2;
        if (x == a[mid]) {
            return mid;
        }
        else if (x < a[mid]) {
            high = mid - 1;
        }
        else {
            low = mid + 1;
        }
    }
    return -1;
}
```

Algorithm 11.6: Binary Search Iterative

```java
public static int binarySearchIter(int[] a, int x) {
    int low = 0;
    int high = a.length - 1; // 2 time units (assignment: 1, subtraction: 1)

    // Loop runs log(n) times and makes 1 comparison per iteration
    while (low <= high) {
        int mid = (low + high) / 2; // For each iteration: 1 addition, 1 division, 1
            assignment (3 time units)

        if (x == a[mid]) { // Comparison to check if element is found (1 time unit per
            iteration)
            return mid; // 1 time unit (return)
        } else if (x < a[mid]) { // Comparison to decide which half to search next (1
            time unit per iteration)
            high = mid - 1; // For each iteration: 1 subtraction, 1 assignment (2 time
                units)
        } else { // If neither condition is satisfied
            low = mid + 1; // For each iteration: 1 addition, 1 assignment (2 time units)
        }
    }
    return -1; // 1 time unit (return)
}
// Detailed Time Complexity Analysis
// ---------------------------------
// Initialisations before the loop: 2 time units (assignments and subtraction)
// For each iteration of the while loop (which runs log(n) times):
// - Loop condition comparison: 1 time unit
// - Midpoint calculation: 3 time units
// - Comparisons to check element: 1 time unit
// - One of the assignments for updating bounds (either low or high): 2 time units
// Total time per iteration: 1 (while comparison) + 3 (mid calculation) + 1 (comparison)
    + 2 (update bounds) = 7 time units per iteration
// The loop runs approximately log(n) times.
// T(n) = 2 (initialisation) + 7 * log(n) (loop iterations) + 1 (final return)
//      = 3 + 7 * log(n)
// Ignoring constants and lower-order terms, we get: T(n) = O(log n)
```

When doing analysis of code, we are mostly interested in the time complexity in terms of Big-$\mathcal{O}$ notation. This allows us to understand how the algorithm scales with input size. The detailed analysis of time units spent on each line of code is useful for understanding the efficiency of the algorithm and identifying bottlenecks. This can be useful when optimizing code for performance. When doing the latter, we still only need to focus on the dominating term in the time complexity, as this is what determines the overall growth rate of the algorithm.

Also, we usually analyse where the time units are spent for each line of code. The following examples use this approach.

Optimising Algorithm Efficiency

The final section of this chapter - and of the book - provides an example of how to optimize an algorithm. We will use the example of finding the sum of all possible pairs in an array to illustrate how to improve the efficiency of an algorithm. The algorithm will be optimized from a naive approach to increasingly more efficient solutions. The algorithm does the following:

- The algorithm calculates the **total sum of all possible pairwise sums** $\text{arr}[i] + \text{arr}[j]$ in the array `arr`.
- This includes every combination of elements added together, considering the order (since both $(\text{arr}[i], \text{arr}[j])$ and $(\text{arr}[j], \text{arr}[i])$ are included).This sum accounts for every combination of elements, including pairs where the same element is paired with itself.

Example 11.18 Here is a manual illustration of the sum of pairs::

Suppose $\text{arr} = [1, 2, 3]$

Manually Calculating All Pair Sums:

$$\begin{aligned}\text{totalSum} &= (1+1) + (1+2) + (1+3) + \\ &\quad (2+1) + (2+2) + (2+3) + \\ &\quad (3+1) + (3+2) + (3+3) \\ &= 2+3+4+3+4+5+4+5+6 \\ &= 36\end{aligned}$$

Let us now consider the **naive approach** to solving this problem which involves iterating through every possible pair in the array:

Algorithm 11.7: Naive Sum of Pairs with Analysis

```java
public static int naiveSumOfPairs(int[] arr) {
    int n = arr.length; // 1 time unit
    int totalSum = 0; // 1 time unit

    // Outer loop over 'i' from 0 to n - 1
    for (int i = 0; i < n; i++) { // 2n + 2
        // Inner loop over 'j' from 0 to n - 1
        for (int j = 0; j < n; j++) { // (2n + 2) * n
            totalSum += arr[i] + arr[j]; // 3n^2
        }
    }
    return totalSum; // 1 time unit
}
// Detailed Time Complexity Analysis
// --------------------------------
// Initializations: 2 time units (n and totalSum assignments)
// Outer loop: 2n + 2 time units
// Inner loop 1: n * (2n + 2) time units
// Inner loop body: 3n^2 time units
```

```
    // Return statement: 1 time unit
    //
    // Total Time: T(n) = 2 (initializations)
    //                  + 2n + 2
    //                  + 2n^2 + 2n
    //                  + 3n^2
    //                  + 1 (return)
    //                  = 5n^2 + 4n + 5
    // Dominating Term: 5n^2
    // Complexity: O(n^2)
```

This algorithm gives us an intituive solution to the problem. Unfortunately, it is very ineffecient for large inputs. Let us now consider a more optimized algorithm for the same problem. Please note that the analysis of the inner loop is complex and inorder to analyse it properly, we refer to Appendix A.

Algorithm 11.8: Optimized Sum of Pairs with Analysis

```java
public static int optimizedSumOfPairs(int[] arr) {
    int n = arr.length;         // 1 time unit
    int totalSum = 0;           // 1 time unit

    // Outer loop over 'i' from 0 to n - 1
    for (int i = 0; i < n; i++) { // 2n + 2 time units
        // Inner loop over 'j' from i to n - 1
        for (int j = i; j < n; j++) { // (2(n - i) + 2) per i, summed over i
            if (i == j) {            // Comparison: (n(n + 1)/2) time units
                totalSum += arr[i] + arr[j]; // 2 time units, executed n times
            } else {
                totalSum += 2 * (arr[i] + arr[j]); // 3 time units, executed (n^2 - n)/2
                    times
            }
        }
    }
    return totalSum; // 1 time unit
}

// Detailed Time Complexity Analysis
// ---------------------------------
// Initializations: 2 time units (n and totalSum assignments)
// Outer loop: 2n + 2 time units (1 initialization, n + 1 comparisons, n increments)
// Inner loop control: Sum of (2(n - i) + 2) for i from 0 to n - 1
//     = Sum of (2n - 2i + 2) from i = 0 to n - 1
//     = 2n^2 + 2n - n(n - 1)
//     = n^2 + 3n time units
// Inner loop body (if condition):
//     - Comparisons: (n(n + 1)/2) time units
//     - if block (i == j): 2 time units, executed n times => 2n
//     - else block (i != j): 3 time units, executed (n^2 - n)/2 times => (3n^2 - 3n)/2
//     Total inner loop body: (n(n + 1)/2) + 2n + (3n^2 - 3n)/2
```

```
    //     = (n^2 + n)/2 + 2n + (3n^2 - 3n)/2
    //     = (4n^2 + 2n)/2
    //     = 2n^2 + n time units
    // Return statement: 1 time unit
    //
    // Total Time: T(n) = 2 (initializations)
    //                  + 2n + 2 (outer loop)
    //                  + n^2 + 3n (inner loop control)
    //                  + 2n^2 + n (inner loop body)
    //                  + 1 (return)
    //                  = 3n^2 + 6n + 5
    // Dominating Term: 3n^2
    // Complexity: O(n^2)
```

While this algorithm is still $\mathcal{O}(n^2)$, it is more efficient than the naive approach, since the dominating term is reduced form $5n^2$ to $3n^2$. To illustrate the practical difference between two algorithms with complexities $5n^2$ and $3n^2$, let's calculate the runtime for $n = 10^5$:

1. NAIVESUMOFPAIRS ($5n^2$):
   $$\text{Runtime} = 5n^2 = 5 \cdot (10^5)^2 = 5 \cdot 10^{10} = 50,000,000,000$$
2. OPTIMIZEDSUMOFPAIRS ($3n^2$):
   $$\text{Runtime} = 3n^2 = 3 \cdot (10^5)^2 = 3 \cdot 10^{10} = 30,000,000,000$$

$$\text{Difference} = 50,000,000,000 - 30,000,000,000 = 20,000,000,000$$

Thus, for $n = 10^5$, NAIVESUMOFPAIRS ($5n^2$) performs **20 billion more operations** than OPTIMIZEDSUMOFPAIRS ($3n^2$). Although both algorithms have the same asymptotic complexity ($\mathcal{O}(n^2)$), the constant factor significantly impacts their performance for large inputs.

The next algorithm is even more efficient, with a time complexity of $\mathcal{O}(n)$.

Algorithm 11.9: Efficient Sum of Pairs with Analysis

```java
public static int efficientSumOfPairs(int[] arr) {
    int n = arr.length;         // 1
    int totalSum = 0;           // 1

    for (int i = 0; i < n; i++) { //2n + 2
        totalSum += 2 * arr[i]; // 3n
    }

    return n * totalSum;        // 2
}
    // Detailed Time Complexity Analysis
    // ---------------------------------
    // Initializations: 2 time units (n and totalSum assignments)
    // Loop: 2n + 2 time units (1 initialization, n + 1 comparisons, n increments)
    // Body: 3n time units (multiplication, addition and assignment, n iterations)
    // Return statement: 2 time units (multiplication and return)
    //
    // Total Time: T(n) = 2 (initializations)
    //                  + 2n + 2 (loop)
    //                  + 3n (body)
    //                  + 2 (return)
    //         = 5n + 6
    //
    // Complexity: O(n)
```

This algorithm is far more efficient than the two previous algorithms. The time complexity is $\mathcal{O}(n)$, which is significantly better than the previous two algorithms. For the input size $n = 10^5$, the runtime is:

1. EFFICIENTSUMOFPAIRS ($5n^2$):
   Runtime $= 5n = 5 \cdot (10^5) = 5 = 50,000$
2. OPTIMIZEDSUMOFPAIRS ($3n^2$):
   Runtime $= 30,000,000,000$
3. Difference $= 30,000,000,000 - 50,000 = 29,999,950,000$

This illustrates how much we can win by going from an algorithm with a time complexity of $\mathcal{O}(n^2)$ to one with $\mathcal{O}(n)$. The difference in the number of operations is staggering. This is why it is important to optimize algorithms to achieve the best performance possible. The following is also an $\mathcal{O}(n)$ algorithm that is even more efficient than the previous one:

Algorithm 11.10: Fast Sum of Pairs with Analysis

```java
public static int fastSumOfPairs(int[] arr) {
    int n = arr.length;            // 1 time unit
    int sumOfArr = 0;              // 1 time unit

    // Loop over 'i' from 0 to n - 1
    for (int i = 0; i < n; i++) {  // 2n + 2 time units (1 initialization, n + 1
        comparisons, n increments)
        sumOfArr += arr[i];        // 2n time units (addition and assignment per
            iteration)
    }

    int totalSum = 2 * n * sumOfArr; // 3 time units (2 multiplications, 1 assignment)
    return totalSum;               // 1 time unit
}
// Updated Detailed Time Complexity Analysis
// ----------------------------------------
// Initializations: 2 time units (n and sumOfArr assignments)
// Loop over 'i':
//    - Loop control: 2n + 2 time units (1 initialization, n + 1 comparisons, n
    increments)
//    - Loop body: 2n time units (sumOfArr += arr[i]; addition and assignment)
// Total for loop: (2n + 2) + 2n = 4n + 2 time units
// Computation of totalSum: 3 time units (2 multiplications, 1 assignment)
// Return statement: 1 time unit
//
```

```
    // Total Time: T(n) = 2 (initializations)
    //                  + 4n + 2 (loop over 'i')
    //                  + 3 (compute totalSum)
    //                  + 1 (return)
    //                  = 4n + 8
    // Dominating Term: 4n
    // Complexity: O(n)
```

As we can see, the fastSumOfPairs algorithm is even more efficient than the previous one and the difference on an input size of $n = 10^5$ is 10,000. We conclude with one final algorithm that - while not more efficient - is considered even better than the previous one.

Algorithm 11.11: Ultra Efficient Sum of Pairs with Analysis

```java
    public static int ultraEfficientSumOfPairs(int[] arr) {
        int n = arr.length;             // 1 time unit
        int sumOfArr = 0;               // 1 time unit

        // Loop over 'value' in 'arr'
        for (int value : arr) {         // 2n + 1 time units (n iterations)
            sumOfArr += value;          // 2n time units (addition and assignment per
                iteration)
        }

        int totalSum = 2 * n * sumOfArr; // 3 time units (2 multiplications, 1 assignment)
        return totalSum;                // 1 time unit
    }

    // Updated Detailed Time Complexity Analysis
    // -----------------------------------------
    // Initializations: 2 time units (n and sumOfArr assignments)
    // Loop over 'arr':
    //    - Loop control: 2n + 1 time units (1 initialization, n comparisons, n increments)
    //    - Loop body: 2n time units (sumOfArr += value; addition and assignment)
    // Total for loop: (2n + 1) + 2n = 4n + 1 time units
    // Computation of totalSum: 3 time units (2 multiplications, 1 assignment)
    // Return statement: 1 time unit
    //
    // Total Time: T(n) = 2 (initializations)
    //                  + (4n + 1) (loop over 'arr')
    //                  + 3 (compute totalSum)
    //                  + 1 (return)
    //                  = 4n + 6
    // Dominating Term: 4n
    // Complexity: O(n)
```

Both have identical asymptotic complexity, and the constant difference is negligible.

Why ultraEfficientSumOfPairs is Better:

1. **Readability:** The enhanced `for-each` loop directly iterates over elements:
   ```
   for (int value :  arr) { sumOfArr += value; }
   ```
   It is concise, avoids indexing, and is less error-prone compared to a traditional `for` loop:
   ```
   for (int i = 0; i < n; i++) { sumOfArr += arr[i]; }
   ```
2. **Maintainability:** Enhanced loops are easier to modify and adapt to different data structures.
3. **Modern Practices:** Aligns with current Java standards, improving clarity and intent.

While both functions perform similarly, ULTRAEFFICIENTSUMOFPAIRS is preferred for its readability, simplicity, and alignment with modern programming practices, making it a more elegant solution.

Bibliography

Cormen, T., Leiserson, C., Rivest, R., & Stein, C. (2022). *Introduction to algorithms* (4th). MIT Press.

Lay, D. (2003). *Linear algebra and its applications*. Pearson Education.

Montgomery, D. (2013). *Applied statistics and probability for engineers, 6th edition*. John Wiley; Sons, Incorporated.

Pishro-Nik, H. (2014). *Introduction to probability, statistics, and random processes*. Kappa Research, LLC.

Rosen, K. H. (2012). *Discrete mathematics and its applications* (7th). McGraw-Hill Education.

Appendix A: Summation

This appendix offers methods for evaluating summations, which occur frequently in the analysis of algorithms. Many of the formulas here appear in any calculus text, but you will find it convenient to have these methods compiled in one place. The content of this appendix is based on the book *Introduction to Algorithms*, (Cormen et al., 2022). The Appendix is concluded with a useful cheat sheet for time complixity analysis.

When an algorithm contains an iterative control construct such as a **while** or **for** loop, you can express its running time as the sum of the times spent on each execution of the body of the loop. For example, in Section 11.3 we argued that the i'th iteration of QUADRATIC-SUM took time proportional to i in the worst case. Adding up the time spent on each iteration produced the summation (or series) $\sum_{i=1}^{n} i$. Evaluating this summation resulted in a bound of $\mathcal{O}(n^2)$ on the worst-case running time of the algorithm. This example illustrates why you should know how to manipulate summations. Before going into details, we provide an overview of the most important summations and their closed forms.

Sum	**Closed Form**
$\sum_{k=0}^{n} ar^k \; (r \neq 1)$	$\frac{ar^{n+1} - a}{r - 1}$
$\sum_{k=1}^{n} k$	$\frac{n(n+1)}{2}$
$\sum_{k=1}^{n} k^2$	$\frac{n(n+1)(2n+1)}{6}$
$\sum_{k=1}^{n} k^3$	$\frac{n^2(n+1)^2}{4}$
$\sum_{k=0}^{n} x^k \; (x \neq 1)$	$\frac{x^{n+1} - 1}{x - 1}$
$\sum_{i=1,2,4,\ldots,n} i$	$2n - 1$
$\sum_{k=0}^{\infty} x^k, \; \lvert x \rvert < 1$	$\frac{1}{1 - x}$
$\sum_{k=1}^{\infty} kx^{k-1}, \; \lvert x \rvert < 1$	$\frac{1}{(1 - x)^2}$
$\sum_{k=1}^{n} \frac{1}{k}$	$\ln n + O(1)$
$\sum_{k=1}^{n} (a + bk)$	$an + b\frac{n(n+1)}{2}$

Table A.1: Some Useful Summation Formulae

Summation Notation

Consider a sequence of numbers $a_1, a_2, \ldots, a_n$, where n is a nonnegative integer. The sum of this sequence, $a_1 + a_2 + \cdots + a_n$, can be represented by the notation $\sum_{k=1}^{n} a_k$. When $n = 0$, this summation is defined to have a value of 0. The result of a finite sum is always well-defined, and the order in which the terms are summed does not affect the final value.

For an infinite sequence $a_1, a_2, \ldots$ of numbers, we represent their infinite sum $a_1 + a_2 + \ldots$ by the notation $\sum_{k=1}^{\infty} a_k$, which corresponds to $\lim_{n\to\infty} \sum_{k=1}^{n} a_k$. If this limit exists, the series is said to converge; otherwise, it diverges. Unlike finite sums, the terms of a convergent series cannot necessarily be rearranged without affecting the outcome. However, in an absolutely convergent series—one where $\sum_{k=1}^{\infty} |a_k|$ also converges — the terms can be reordered without changing the sum.

$$\sum_{i=i_0}^{n} ca_i = c \sum_{i=i_0}^{n} a_i$$

where c is any number. So, we can factor constants out of a summation.

Similarly, we can split a summation into two summations:

$$\sum_{i=i_0}^{n} (a_i \pm b_i) = \sum_{i=i_0}^{n} a_i \pm \sum_{i=i_0}^{n} b_i$$

Note that we started the series at i_0 to denote the fact that they can start at any value of i that we need them to. Also note that while we can break up sums and differences as we did above we can't do the same thing for products and quotients. In other words,

$$\sum_{i=i_0}^{n} (a_i b_i) \neq \left(\sum_{i=i_0}^{n} a_i\right)\left(\sum_{i=i_0}^{n} b_i\right) \qquad \text{and} \qquad \sum_{i=i_0}^{n} \frac{a_i}{b_i} \neq \frac{\sum_{i=i_0}^{n} a_i}{\sum_{i=i_0}^{n} b_i}.$$

Linearity of Summation

For any real number c and any finite sequences $a_1, a_2, \ldots, a_n$ and $b_1, b_2, \ldots, b_n$,

$$\sum_{k=1}^{n} (ca_k + b_k) = c\sum_{k=1}^{n} a_k + \sum_{k=1}^{n} b_k.$$

The linearity property also applies to infinite convergent series.

The linearity property applies to summations incorporating asymptotic notation. For example,

$$\sum_{k=1}^{n} \Theta(f(k)) = \Theta\left(\sum_{k=1}^{n} f(k)\right).$$

In this equation, the Θ-notation on the left-hand side applies to the variable k, but on the right-hand side, it applies to n. Such manipulations also apply to infinite convergent series.

Arithmetic Series

The summation

$$\sum_{k=1}^{n} k = 1 + 2 + \cdots + n$$

is an arithmetic series and has the value

$$\sum_{k=1}^{n} k = \frac{n(n+1)}{2} \tag{A.1}$$

$$= \Theta\left(n^2\right). \tag{A.2}$$

A general arithmetic series includes an additive constant $a \geq 0$ and a constant coefficient $b > 0$ in each term, but has the same total asymptotically:

$$\sum_{k=1}^{n} (a + bk) = \Theta\left(n^2\right). \tag{A.3}$$

Sums of Squares and Cubes

The following formulas apply to summations of squares and cubes:

$$\sum_{k=0}^{n} k^2 = \frac{n(n+1)(2n+1)}{6}, \tag{A.4}$$

$$\sum_{k=0}^{n} k^3 = \frac{n^2(n+1)^2}{4}. \tag{A.5}$$

Geometric Series

For real $x \neq 1$, the summation

$$\sum_{k=0}^{n} x^k = 1 + x + x^2 + \cdots + x^n$$

is a geometric series and has the value

$$\sum_{k=0}^{n} x^k = \frac{x^{n+1} - 1}{x - 1}. \tag{A.6}$$

The infinite decreasing geometric series occurs when the summation is infinite and $|x| < 1$:

$$\sum_{k=0}^{\infty} x^k = \frac{1}{1 - x}. \tag{A.7}$$

If we assume that $0^0 = 1$, these formulas apply even when $x = 0$.

A **special** case of the geometric series arises when the terms form a progression of powers of 2. That is, the terms are $1, 2, 4, 8, \ldots, n$. In this case:

$$\sum_{i=1,2,4,\ldots,n} i = 1 + 2 + 4 + 8 + \cdots + n.$$

This series can be derived from the general formula for a geometric series. The general form of a geometric series is:

$$S = a + ar + ar^2 + \cdots + ar^{k-1},$$

where:

- a is the first term ($a = 1$ in this case),
- r is the common ratio ($r = 2$ here),
- k is the number of terms in the series.

The number of terms, k, is determined by how many times the progression $1, 2, 4, \ldots, n$ doubles until reaching n. Since $n = 2^{k-1}$, we have $k = \log_2(n) + 1$.

Using the geometric series formula (A.6):

$$\sum_{i=1,2,4,\ldots,n} i = \frac{2^k - 1}{2 - 1} = 2^k - 1.$$

Substituting $k = \log_2(n) + 1$:

$$\sum_{i=1,2,4,\ldots,n} i = 2^{\log_2(n)+1} - 1 = 2n - 1.$$

Thus, the sum of the powers of 2 up to n is:

$$\sum_{i=1,2,4,\ldots,n} i = 2n - 1.$$

Harmonic Series

For positive integers n, the nth harmonic number is

$$\begin{aligned} H_n &= 1 + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \cdots + \frac{1}{n} \\ &= \sum_{k=1}^{n} \frac{1}{k} && \text{(A.8)} \\ &= \ln n + O(1). && \text{(A.9)} \end{aligned}$$

Integrating and Differentiating

Integrating or differentiating the formulas above yields additional formulas. For example, differentiating both sides of the infinite geometric series (A.7) and multiplying by x gives

$$\sum_{k=0}^{\infty} kx^k = \frac{x}{(1-x)^2} \qquad \text{(A.10)}$$

for $|x| < 1$.

Telescoping Series

For any sequence $a_0, a_1, \ldots, a_n$

$$\sum_{k=1}^{n} (a_k - a_{k-1}) = a_n - a_0, \tag{A.11}$$

since each of the terms $a_1, a_2, \ldots, a_{n-1}$ is added in exactly once and subtracted out exactly once. We say that the sum telescopes. Similarly,

$$\sum_{k=0}^{n-1} (a_k - a_{k+1}) = a_0 - a_n.$$

As an example of a telescoping sum, consider the series

$$\sum_{k=1}^{n-1} \frac{1}{k(k+1)}$$

Rewriting each term as

$$\frac{1}{k(k+1)} = \frac{1}{k} - \frac{1}{k+1},$$

gives

$$\begin{aligned} \sum_{k-1}^{n-1} \frac{1}{k(k+1)} &= \sum_{k-1}^{n-1} \left(\frac{1}{k} - \frac{1}{k+1} \right) \\ &= 1 - \frac{1}{n}. \end{aligned} \tag{A.12}$$

Reindexing Sums

A series can sometimes be simplified by changing its index, often reversing the order of summation. Consider the series $\sum_{k=0}^{n} a_{n-k}$. Because the terms in this summation are $a_n, a_{n-1}, \ldots, a_0$, we can reverse the order of indices by letting $j = n - k$ and rewrite this summation as

$$\sum_{k=0}^{n} a_{n-k} = \sum_{l=0}^{n} a_j$$

Generally, if the summation index appears in the body of the sum with a minus sign, it's worth thinking about reindexing.

As an example, consider the summation

$$\sum_{k=1}^{n} \frac{1}{n-k+1}. \tag{A.13}$$

The index k appears with a negative sign in $\frac{1}{(n-k+1)}$. And indeed, we can simplify this summation, this time setting $j = n - k + 1$, yielding

$$\sum_{k=1}^{n} \frac{1}{n-k+1} = \sum_{i=1}^{n} \frac{1}{j}, \tag{A.14}$$

which is just the harmonic series (A.8).

Products

The finite product $a_1 a_2 \ldots a_n$ can be expressed as

$$\prod_{k=1}^{n} a_k .$$

If $n = 0$, the value of the product is defined to be 1 . You can convert a formula with a product to a formula with a summation by using the identity

$$\log \left(\prod_{k=1}^{n} a_k \right) = \sum_{k=1}^{n} \log a_k .$$

Cheat Sheet for Time Complexity Analysis

This section provides essential formulas and shortcuts for asymptotic analysis and algorithm evaluation. These complement the detailed summation methods in this appendix.

- **Logarithmic Factorial Approximation (Stirling's Approximation):**
  $$\log(n!) \approx n \log n - n.$$
- **Logarithmic Exponent Rule:**
  $$2^{\log_2 n} = n.$$
- **Base as a Power with Logarithmic Exponent:** If the base is a^b and the exponent involves $\log_a$, then:
  $$(a^b)^{\log_a(n^c)} = n^{bc}.$$
  Example: $4^{\log_2(n^2)} = n^4$ (since $4 = 2^2$).
- **Power of a Logarithm Rule:**
  $$\log(n^k) = k \log n, \quad \text{for } k > 0.$$
- **Logarithmic Growth for Powers of 2:** For $n = 2^k$,
  $$k = \log_2 n.$$
- **Simplifying Logarithmic Expressions:**
  $$\log_a n = \frac{\log_b n}{\log_b a}, \quad \text{for } a, b > 0.$$
- **Common Asymptotic Comparisons:**
  $$n^c \ll c^n \quad \text{for any constant } c > 1.$$
- **Factorial Asymptotics (Expanded Stirling's Approximation):** Using Stirling's formula, the growth of $n!$ is approximately:
  $$n! = \Theta\left(\sqrt{2\pi n} \left(\frac{n}{e} \right)^n \right).$$
- **Sum of Powers of 2:** If the terms in a series form powers of 2 (e.g., $1, 2, 4, 8, \ldots, n$), the sum is:
  $$\sum_{i=1,2,4,\ldots,n} i = 2n - 1.$$